The Group Cumulative Scheduling Problem

Lucas Groleaz

▶ To cite this version:

Lucas Groleaz. The Group Cumulative Scheduling Problem. Computer Science [cs]. Institut National des Sciences Appliquées de Lyon, 2021. English. NNT : . tel-03266690v1

HAL Id: tel-03266690

https://hal.science/tel-03266690v1

Submitted on 22 Jun 2021 (v1), last revised 26 Oct 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre :

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée au sein de l'INSA de Lyon

École doctorale n° 512

InfoMaths

Spécialité de doctorat :

Informatique

Soutenue publiquement le 07-06-2021, par

Lucas Groleaz

The Group Cumulative Scheduling Problem

Devant le jury composé de :

M. Christian Artigues	Directeur de Recherche	CNRS	Rapporteur
M. Yves Deville	Professeur des Universités	Université Catholique de Louvain	Rapporteur
Mme Nadia Brauner	Professeure des Universités	Université Grenoble Alpes	Examinatrice
M. Philippe Laborie	Docteur en Informatique	IBM	Examinateur
Mme Christine Solnon	Professeure des Universités	INSA-LYON	Directrice de thèse
M. Samba Ndojh Ndiaye	Maître de conférence	Universités LYON 1	Co-directeur de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **https://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr	**M. Stéphane DANIELE** C2P2-CPE LYON-UMR 5265 Bâtiment F308, BP 2077 43 Boulevard du 11 novembre 1918 69616 Villeurbanne **directeur@edchimie-lyon.fr**
E.E.A.	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **https://edeea.universite-lyon.fr** Sec. : Stéphanie CAUVIN Bâtiment Direction INSA Lyon Tél : 04.72.43.71.70 secretariat.edeea@insa-lyon.fr	**M. Philippe DELACHARTRE** INSA LYON Laboratoire CREATIS Bâtiment Blaise Pascal, 7 avenue Jean Capelle 69621 Villeurbanne CEDEX Tél : 04.72.43.88.63 **philippe.delachartre@insa-lyon.fr**
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http://e2m2.universite-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.e2m2@univ-lyon1.fr	**M. Philippe NORMAND** Université Claude Bernard Lyon 1 UMR 5557 Lab. d'Ecologie Microbienne Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX **philippe.normand@univ-lyon1.fr**
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http://ediss.universite-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.ediss@univ-lyon1.fr	**Mme Sylvie RICARD-BLUM** Institut de Chimie et Biochimie Moléculaires et Supramoléculaires (ICBMS) - UMR 5246 CNRS - Université Lyon 1 Bâtiment Raulin - 2ème étage Nord 43 Boulevard du 11 novembre 1918 69622 Villeurbanne Cedex Tél : +33(0)4 72 44 82 32 **sylvie.ricard-blum@univ-lyon1.fr**
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 infomaths@univ-lyon1.fr	**M. Hamamache KHEDDOUCI** Université Claude Bernard Lyon 1 Bât. Nautibus 43, Boulevard du 11 novembre 1918 69 622 Villeurbanne Cedex France Tél : 04.72.44.83.69 **hamamache.kheddouci@univ-lyon1.fr**
Matériaux	**MATÉRIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec. : Yann DE ORDENANA Tél : 04.72.18.62.44 yann.de-ordenana@ec-lyon.fr	**M. Stéphane BENAYOUN** Ecole Centrale de Lyon Laboratoire LTDS 36 avenue Guy de Collongue 69134 Ecully CEDEX Tél : 04.72.18.64.37 **stephane.benayoun@ec-lyon.fr**
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http://edmega.universite-lyon.fr** Sec. : Stéphanie CAUVIN Tél : 04.72.43.71.70 Bâtiment Direction INSA Lyon mega@insa-lyon.fr	**M. Jocelyn BONJOUR** INSA Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69621 Villeurbanne CEDEX **jocelyn.bonjour@insa-lyon.fr**
ScSo	**ScSo*** **https://edsciencessociales.universite-lyon.fr** Sec. : Mélina FAVETON INSA : J.Y. TOUSSAINT Tél : 04.78.69.77.79 melina.faveton@univ-lyon2.fr	**M. Christian MONTES** Université Lumière Lyon 2 86 Rue Pasteur 69365 Lyon CEDEX 07 **christian.montes@univ-lyon2.fr**

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Contents

Résumé en français 11

Introduction 15

Notations 17

I Background on Scheduling 19

1 Definitions 21
1.1 Constrained Optimization Problems . . . 21
1.2 Scheduling problems . . . 23
1.2.1 Machine environment α . . . 24
Single Machine Models $\alpha = 1$ . . . 24
Parallel machine models . . . 25
Machine in series - Multi-stage models - Shop problems . . . 25
1.2.2 Constraints and job characteristics : field β . . . 27
1.2.3 Objective functions : field γ . . . 28
1.2.4 RCPSP . . . 28
1.2.5 Thesis scope . . . 29
Assignment problem $P||C_{max}$ . . . 29
Release date and due date $P|r_j|\sum T_j$ . . . 30
Speed problem $Q|r_j|\sum T_j$ . . . 30
Sequence dependent setup times problem $Q|r_j, s_{jk}|\sum T_j$ . . . 31
Breaks problem $Q|r_j, brkdwn|\sum T_j$ . . . 31
Breaks and sequence-dependent setup times problem $Q|r_j, brkdwn, s_{jk}|\sum T_j$ . . . 32
1.3 Computational Complexity . . . 32
1.3.1 Decision problems . . . 33
1.3.2 Complexity classes . . . 33
1.3.3 Problem reduction . . . 34
1.4 Discussion . . . 34

2 Main solving approaches 37
2.1 Greedy Approaches . . . 38
2.1.1 Dispatching rules . . . 38
2.1.2 Greedy Randomized Approaches . . . 39
2.2 Ant Colony Optimization (ACO) . . . 40
2.3 Local Search . . . 42
2.3.1 Construction of an initial solution . . . 42
2.3.2 Neighborhood . . . 42

2.3.3 Choosing a neighbor in the neighborhood . . . 44
2.4 Linear Programming . . . 45
2.4.1 Mixed Integer Linear Programming . . . 47
2.4.2 Models for scheduling problems . . . 47
Assignment problem $P||C_{max}$ . . . 47
Release date and due date : $P|r_j|\sum T_j$ . . . 47
Speeds : $Q|r_j|\sum T_j$ . . . 49
Sequence dependent setup times : $Q|r_j, s_{jk}|\sum T_j$ . . . 50
2.5 Constraint Programming . . . 50
2.5.1 Type of data and constraint . . . 50
2.5.2 Constraint propagation . . . 51
2.5.3 Branch and propagate . . . 51
2.5.4 Model and solvers . . . 52
2.5.5 Models for scheduling problems . . . 52
Scheduling variables and constraints . . . 52
Assignment problem $P||C_{max}$: . . . 53
Release date and due date $P|r_j|\sum T_j$: . . . 54
Speed problem $Q|r_j|\sum T_j$: . . . 54
Sequence dependent setup times problem $Q|r_j, s_{jk}|\sum T_j$: . . . 54
Breaks problem $Q|r_j, brkdwn|\sum T_j$: . . . 54
Breaks and sequence-dependent setup times problem $Q|r_j, brkdwn, s_{jk}|\sum T_j$: . . . 55
2.6 Discussion . . . 56

II The GCSP 59

3 Study of the GCSP 61
3.1 Definition . . . 61
3.2 Related problems . . . 63
3.2.1 Cumulative scheduling problems . . . 63
3.2.2 Open Stack Problems . . . 63
Definition and example . . . 63
Link with the GCSP . . . 64
3.2.3 Hierarchical scheduling problems . . . 64
3.3 Complexity . . . 66
3.3.1 Classical cumulative constraints . . . 66
3.3.2 Group cumulative constraint . . . 67
3.4 Discussion . . . 71

4 New approaches for solving the GCSP 73
4.1 Model for Constraint Programming . . . 73
4.2 Adapting solving methods to the *GCSP* . . . 74
4.2.1 Local search . . . 74
4.2.2 Greedy constructions . . . 76
4.2.3 Ant Colony Optimization . . . 80
4.3 New Hybrid CPO-ACO approach . . . 81
4.4 Discussion . . . 82

III Experimental Evaluation on a New Benchmark **83**

5 Description of the Benchmark **85**
5.1 Description of our applicative context . . . 85
5.1.1 Scheduling applications . . . 85
5.1.2 ERP . . . 86
5.1.3 Infologic and Copilote . . . 86
5.2 Features of the data-set . . . 87
5.3 Discussion . . . 89

6 Experimental Results **91**
6.1 Experimental settings . . . 91
6.1.1 Algorithms and parameters' choice . . . 92
Tabu search . . . 92
ACO . . . 93
Constraint Programming . . . 94
MIP . . . 94
Combining ACO and another approach (CPO-ACO and ACO-Tabu) . . . 94
6.1.2 Performance measures . . . 95
6.2 Individual problems' results . . . 96
6.2.1 Assignment problem $P||C_{max}$ . . . 98
6.2.2 Release date and due date : $P|r_j|\sum T_j$ . . . 100
6.2.3 Speeds : $Q|r_j|\sum T_j$ . . . 102
6.2.4 Sequence dependent setup times : $Q|r_j, s_{jk}|\sum T_j$ . . . 104
6.2.5 Breaks : $Q|r_j, brkdwn|\sum T_j$ . . . 106
6.2.6 Breaks and setup times: $Q|r_j, brkdwn, s_{jk}|\sum T_j$ . . . 108
6.2.7 GC_{loose} no breaks: $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$ . . . 110
6.2.8 GC_{tight} no breaks: $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$ . . . 112
6.2.9 GC_{loose} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$ . . . 114
6.2.10 GC_{tight} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$ . . . 116
6.3 Results over all problems . . . 117
6.4 Discussion . . . 121

7 Automatic Algorithm Selection **123**
7.1 Definition and goals of automatic algorithm selection . . . 123
7.2 Performance measure . . . 124
7.3 Instance features . . . 125
7.4 Choice of an Approach for Learning the Mapping Model . . . 127
7.5 Llama results . . . 128
7.6 Discussion . . . 130

8 Experimental Evaluation for Dynamic Scheduling **133**
8.1 Motivation and definition of the problem . . . 133
8.2 Review of dynamic scheduling problems . . . 133
8.3 Objectives and framework . . . 135
8.4 Experimental results . . . 138
8.5 Towards anticipation . . . 140
8.6 Discussion . . . 141

9 Conclusion **143**

Remerciements

Même s'il n'y a qu'un seul nom en face de la thèse, celle-ci est avant tout un travail collectif. Ce qui fait une thèse, ce sont avant tout des rencontres et des échanges. J'ai bien quelques idées quand je suis seul, mais c'est le fait de discuter de celles-ci, d'écouter celles des autres, de prendre en compte les remarques qui permet de se dépasser et d'aboutir à des résultats plus concluants. Je ne saurais que trop répéter ce si juste proverbe : "Seul on va plus vite, ensemble on va plus loin". J'aimerais remercier toutes les personnes qui ont contribué à cette thèse, de près ou de loin.

Merci tout d'abord à Christine, ma directrice de thèse, pour tes remarques pertinentes, tes idées foisonnantes et ton exigence. Merci pour ta profonde implication dans les moments décisifs de cette thèse, pour tes relectures et tes réécritures. Merci aussi pour ces discussions, qui n'ont pas forcément de rapports avec la thèse, mais qui incitent sans cesse à la réflexion.
Merci à Samba, mon co-directeur de thèse, pour ton implication, malgré les difficultés, pour ton soutien, et pour tes suggestions avisées.

Je remercie Messieurs Christian Artigues et Yves Deville qui m'ont fait l'honneur d'être rapporteurs de cette thèse, et je les remercie également pour leurs pertinents rapports. Je tiens également à remercier Monsieur Philippe Laborie et Madame Nadia Brauner d'avoir accepté d'être examinateur de cette thèse.

Merci à Infologic, et plus particulièrement à André Chabert, pour cette confiance sans cesse renouvelée, pour ces visions de l'entreprise et de la recherche. Cette volonté de rapprocher recherche et problématiques industrielles est une vraie force. Merci pour ce goût du terrain, et merci de partager cette envie constante de toujours résoudre les problématiques rencontrées par les utilisateurs de l'ERP.
Au sein d'Infologic, je voudrais également remercier Erwan, pour tes remarques toujours très justes, et tes connaissances infaillibles à la fois sur Copilote et à la fois sur le fonctionnement des différents sites des clients.
Merci également à Martin, pour tous les échanges que nous avons pu avoir, qu'ils soient orientés recherche opérationnelle, écologie, jardin ou autres. Tes connaissances en recherche opérationnelle, tes remarques toujours pertinentes, et tes idées m'ont beaucoup aidées dans cette thèse. Travailler à tes côtés a toujours été un plaisir et je t'en remercie.
Merci également à Maxime, pour m'avoir partagé ton vécu de thèse, pour tes conseils quant à la gestion de la mienne, et merci pour ta thèse dont certains passages de la mienne sont librement inspirés.
Merci plus généralement à toute l'agence de Lyon, à ceux qui sont encore là et à ceux qui sont partis, pour ces repas, ces discussions, ces bières, ces moments de sport (foot, cross-fit et autres volley), ...

Je voudrais également remercier mes parents pour leur soutien, leurs encouragements et pour l'éducation qu'ils m'ont donnée. Je suis très heureux et fier de cette thèse aujourd'hui, c'est un bel accomplissement professionnel, mais mon plus grand souhait sera toujours d'essayer d'être un jour pour mes enfants ce que vous êtes pour moi. Vous êtes et vous serez toujours un modèle pour moi. Encore merci pour tout ce que vous avez fait pour moi, tout au long de ma vie.

Merci à mes sœurs, Marie et Lisa, pour tous les moments que nous avons partagés, vous êtes une source de motivation incroyable, et essayer de vous rendre fier m'animera toujours.
Merci également à mon frère Fabien, qui veille de tout la-haut, et qui m'inspire du plus profond de moi-même. Ton omniprésence est un moteur, simplement ralenti par la douleur du manque.
Merci plus généralement à ma famille, toutes ces années passées à vos côtés ont fait de moi ce que je suis aujourd'hui, et m'ont permis de grandir dans un cadre de vie sain et protecteur.

Merci, bien évidemment, à ma compagne Joanna. Nous partageons tant de choses, depuis tant d'années, merci pour tous ces moments. Et merci pour tous les moments que nous allons encore partager. Je te remercie de toujours me soutenir, même dans les moments les plus difficiles, de toujours me faire rire, d'être ma complice en toute occasion, et de toujours tout donner pour moi. Je suis fier d'avancer à tes côtés.
Merci à ma belle-famille pour votre soutien et votre générosité débordante.

Merci à mes deux petits garçons, Eden et Lenny, vous avez su me réveiller les nuits où je ne me réveillais pas pour penser à la thèse. Mais surtout merci pour ce bonheur inconditionnel que vous m'apportez. Quelque soit l'avancement de ma thèse, pandémie ou non, vous êtes toujours là pour m'inonder de joie. Vous êtes ma plus grande fierté.

Finalement je voudrais remercier tous mes amis. Que ce soit mes amis de Diemoz, de Paris, de Genas, de prépa, de sport ou d'ailleurs, merci à tous d'être dans ma vie. Merci pour vos encouragements, pour toutes ces soirées, ces bières, ces voyages, ces trails, ces parties de cartes, de foot, de ping-pong, de pétanque, ou de Schifumi. Merci pour tous ces moments partagés et pour cet équilibre que vous m'avez apporté pendant ces années de thèse.

Résumé en français

Motivation

Un ERP (Enterprise Resource Planning) ou encore parfois appelé PGI (Progiciel de Gestion Intégré) est un système d'information qui permet de gérer et suivre au quotidien l'ensemble des informations et des services opérationnels d'une entreprise. Développé par un éditeur unique, l'ERP est ensuite déployé sur différents sites industriels. Il doit ainsi être hautement paramétrable afin de pouvoir s'adapter aux différentes situations auxquelles il est confronté. Les ERP sont souvent composés de différents modules, qui répondent à des besoins métiers différents.
La société Infologic développe un ERP, appelé Copilote, spécialisé pour les entreprises du secteur agro-alimentaire. Il intègre notamment un module permettant d'ordonnancer les opérations de production dans les ateliers, ainsi qu'un module permettant d'ordonnancer les opérations de préparation de commandes. Ces modules doivent donc apporter des solutions à différents problèmes d'ordonnancement. De plus, comme chaque entrepôt utilisant ces modules a un fonctionnement différent, les problèmes d'ordonnancement à traiter ont des contraintes et des objectifs différents.

La problématique est donc d'avoir une méthode qui puisse traiter les différents problèmes d'ordonnancement rencontrés par Copilote, tout en apportant des solutions de la meilleure qualité possible. Les ERP étant hautement configurables, il est possible d'envisager une stratégie où les méthodes de résolutions pour les problèmes d'ordonnancement sont choisies par l'installateur de l'ERP ou par l'utilisateur lui-même. Cependant, une telle stratégie nécessite que les utilisateurs aient des connaissances expertes, à la fois sur la taxonomie des problèmes d'ordonnancement et sur les méthodes de résolution utilisées. En pratique ces deux conditions sont rarement réunies, ce qui motive l'idée d'avoir une méthode qui sache s'adapter automatiquement aux problèmes qu'elle rencontre.

Par ailleurs, bien que la littérature concernant les problèmes d'ordonnancement soit vaste, une contrainte particulière rencontrée par les utilisateurs de *Copilote* ne peut que difficilement être modélisée en utilisant les éléments connus de la littérature. Cette contrainte modélise la réalité suivante : lors du processus de préparation de commandes, les opérateurs préparent différents produits pour réaliser chaque commande. Cependant sitôt qu'un produit d'une commande est prêt, il doit être positionné sur une palette. La palette reste ouverte jusqu'à ce que tous les produits de la commande aient été préparés. Par ailleurs, cette palette occupe de la place sur le sol, et cette place est limitée. La contrainte impose donc que le nombre de palettes qui sont simultanément ouvertes ne dépasse pas une certaine quantité.

Finalement, les commandes que traitent les préparateurs de commandes arrivent tout au long de la journée. Ainsi, il n'est pas envisageable de calculer un ordonnancement des tâches le matin qui serait suivi toute la journée. Les algorithmes utilisés doivent donc être capables de réagir à l'arrivée de nouvelles commandes afin de positionner leur préparation dans le calendrier des préparateurs. Néanmoins, les préparateurs ayant connaissance des tâches qui leur sont assignées, perturber sans cesse leur planning est source d'inconfort dans leur travail. Ainsi il est nécessaire d'essayer de perturber le moins possible leur planning, tout en essayant de préparer un maximum de commandes dans les temps.

Objectifs et contributions

Un premier objectif de cette thèse consiste à modéliser et étudier la contrainte évoquée ci-dessus limitant le nombre de commandes qui peuvent être traitées en simultané. Nous montrons en particulier dans cette thèse, que si l'on connaît les tâches que chaque préparateur de commandes doit effectuer, et dans quel ordre il doit les effectuer, alors trouver une

date de début et une date de fin pour chacune de ces tâches telles que cette contrainte soit respectée est un problème $NP-complet$. Nous examinons notamment l'impact de ce résultat sur les différents algorithmes d'ordonnancement existant dans la littérature. Nous introduisons également un nouvel algorithme, combinaison de deux algorithmes existants, et nous montrons qu'il est particulièrement bien adapté à ce problème d'ordonnancement.

Par ailleurs nous évaluons expérimentalement plusieurs algorithmes d'ordonnancement de l'état de l'art sur un jeu de données fourni par Infologic. En considérant différentes contraintes et différentes fonctions objectif, nous évaluons leur impact sur les algorithmes utilisés. En particulier, cela nous permet d'identifier quels algorithmes sont efficaces pour traiter les différentes contraintes. Nous constatons en effet qu'aucun algorithme est meilleur que tous les autres pour toutes les contraintes.

Comme mentionné ci-dessus, *Copilote* doit être capable de gérer le mieux possible les différents problèmes d'ordonnancement qu'il rencontre. Cependant aucun algorithme n'est meilleur que tous les autres sur tous les problèmes, et, par ailleurs, les utilisateurs n'ont souvent pas l'expertise nécessaire pour choisir la méthode la plus adaptée au problème qu'ils rencontrent. Pour cette raison nous étudions l'intérêt d'utiliser une méthode de sélection automatique d'algorithme, qui choisit en fonction des caractéristiques du problème à traiter la méthode de résolution la plus adaptée.

Nous évaluons également l'impact sur la qualité des solutions trouvées du fait de ne pas modifier les tâches assignées aux différents préparateurs de commandes. Plus précisément, à chaque arrivée d'une nouvelle commande, nous interdisons de modifier les ϵ prochaines minutes du planning des opérateurs. Nous évaluons dans cette thèse l'influence de ce paramètre ϵ.

Plan de la thèse

La première partie de cette thèse vise à la positionner au sein de l'état de l'art. En particulier, dans le chapitre 1 nous introduisons quelques éléments concernant les problèmes d'optimisation sous contraintes ainsi que quelques éléments de la théorie de la complexité. Nous présentons également une taxonomie des problèmes d'ordonnancement et situons les problèmes d'ordonnancement étudiés dans cette thèse au sein de cette taxonomie. Dans le chapitre 2 nous présentons et illustrons certaines des principales méthodes utilisées dans la littérature pour résoudre ces problèmes.

Dans la deuxième partie de la thèse nous introduisons et étudions une nouvelle contrainte pour les problèmes d'ordonnancement (nommée *contrainte cumulative de groupe (GC)*), que nous utilisons pour modéliser le problème, précédemment évoqué, de place disponible pour traiter plusieurs commandes en parallèle. Dans le chapitre 3 nous introduisons formellement cette contrainte et nous étudions ses liens avec des problèmes proches présents dans la littérature. Nous donnons également la preuve de *NP*-complétude mentionnée précédemment et nous étudions son impact sur les méthodes de résolution existantes. En particulier, dans le chapitre 4 nous proposons différentes approches pour adapter ces méthodes de résolution afin qu'elles puissent prendre en compte cette nouvelle contrainte.

Dans la dernière partie de cette thèse, nous évaluons expérimentalement les différentes méthodes sus-mentionnées sur le jeu de données fournis par Infologic. En particulier, dans le chapitre 5 nous décrivons de manière précise notre de jeu de données ainsi que notre contexte industriel. Nous évoquons également plusieurs articles évaluant l'intérêt de la mise en place de techniques d'ordonnancement au sein de l'industrie. Dans le chapitre 6 nous donnons les résultats de l'évaluation expérimentale des méthodes utilisées sur notre jeu de données. En particulier, nous évaluons l'impact des différentes contraintes utilisées dans les différents problèmes d'ordonnancement sur les méthodes présentées. Dans le chapitre 7 nous introduisons et évaluons l'intérêt d'utiliser la méthode de *sélection automatique d'algorithmes*. Cette méthode utilise des techniques de *machine learning* pour choisir automatiquement quel algorithme utiliser en fonction des caractéristiques de l'instance à résoudre. Finalement, dans le chapitre 8 nous étudions un problème d'ordonnancement en prenant en compte son aspect dynamique. En particulier, les différentes commandes à réaliser ne sont pas toutes connues à l'avance et sont révélées à mesure que l'algorithme s'exécute. Après une courte présentation des techniques utilisées dans la littérature pour traiter les problèmes dynamiques, nous évaluons l'impact lié au fait d'interdire de modifier les futures tâches que doivent réaliser les opérateurs.

Publications

La contrainte *GC* a été introduite dans un article publié à la conférence *The Genetic and Evolutionary Computation Conference (GECCO)* [GNS20b] en 2020, avec la sélection automatique de paramètres pour les algorithmes de colonies de fourmis (une partie du chapitre 7). L'étude théorique de la contrainte *GC* (notamment la preuve de $\mathcal{NP}-$Complétude), ainsi que l'algorithme combinant *CPO* et *ACO* ont été publiés à la conférence *international conference on principles and practice of CP* en 2020 [GNS20a].
Nous prévoyons de publier dans un article de journal, les résultats expérimentaux du chapitre 6 qui évaluent l'impact des différentes contraintes sur les différents algorithmes. Nous prévoyons également d'y inclure les adaptations considérant la contrainte *GC* décrites au chapitre 4, ainsi que l'utilisation de la *sélection automatique d'algorithme* sur le jeu de données complet (chapitre 7).

Introduction

Motivation

An ERP (Enterprise Resource Planning) is an information system that allows to manage and monitor on a daily basis, all the information and operational services of a company. Developed by a single editor, the ERP is then deployed on different industrial sites. It must therefore be highly configurable in order to be able to adapt to the different situations it faces. ERPs are often composed of different modules, which meet different business needs.
The company Infologic develops an ERP, called *Copilote*, specialized for companies in the food industry. In particular, it integrates a module for scheduling production operations in the workshops, as well as a module for scheduling order preparation operations. These modules must therefore provide solutions to various scheduling problems. In addition, since each warehouse using these modules operates differently, the scheduling problems to be dealt with have different constraints and objectives.

The problem is therefore to have a method that can handle the different scheduling problems encountered by *Copilote*, while providing the best possible solutions. Since ERPs are highly configurable, it is possible to envisage a strategy where the methods for solving scheduling problems are chosen by the ERP installer or by the user himself. However, such a strategy requires users to have expert knowledge on both the taxonomy of scheduling problems and the resolution methods used. In practice, these two conditions are rarely met, which motivates the idea of having a method that can automatically adapt to the problems encountered.

Moreover, although the literature on scheduling problems is vast, a particular constraint encountered by users of *Copilote* is difficult to model using known elements of the literature. This constraint models the following reality: during the order picking process, operators prepare different products to fulfill each order. However, as soon as a product in an order is ready, it must be positioned on a pallet. The pallet remains open until all products in the order have been prepared. In addition, this pallet takes up space on the floor, and this space is limited. The constraint therefore requires that the number of pallets that are simultaneously opened does not exceed a certain quantity.

Finally, the orders processed by the order pickers arrive throughout the day. Thus, it is not possible to calculate a task schedule in the morning that would be followed throughout the day. The algorithms used must therefore be capable of reacting to the arrival of new orders in order to position their preparation in the pickers' schedule. Nevertheless, since the pickers are aware of the tasks assigned to them, constantly disrupting their schedule is a source of discomfort in their work. It is therefore necessary to try to disrupt their schedule as little as possible, while trying to prepare a maximum number of orders in time.

Goals and contributions

A first objective of this thesis is to model and study the above-mentioned constraint limiting the number of orders that can be processed simultaneously. We show in particular in this thesis, that if we know the tasks that each order picker must perform, and in what order he must perform them, then finding a start date and an end date for each of these tasks such that this constraint is respected is $NP-complete$ problem. We examine in particular the impact of this result on the different scheduling algorithms existing in the literature. We also introduce a new algorithm, a combination of two existing algorithms, and show that it is particularly well adapted to this scheduling problem.

In addition, we experimentally evaluate several state of the art scheduling algorithms on a data set provided by Infologic. By considering different constraints and objective functions, we evaluate their impact on the algorithms used. In particular, this allows us to identify which algorithms are efficient in dealing with the different constraints. Indeed, we find that no algorithm is better than all others for all constraints.

As mentioned above, *Copilote* must be able to handle as well as possible the different scheduling problems it encounters. However, no algorithm is better than all others on all problems, and users often do not have the expertise to choose the best method for the problem they encounter. For this reason we study this interest of using a method of automatic algorithm selection, which chooses according to the characteristics of the problem to be treated the most suitable method of resolution.

We also evaluate the impact on the quality of the solutions found by not modifying the tasks assigned to the different order pickers. More precisely, each time a new order arrives, we forbid to modify the next x minutes of the operators' schedule. We evaluate in this thesis the influence of this x parameter.

Outline of the thesis

The first part of this thesis aims to place it within the state of the art. In particular, in Chapter 1 we introduce some elements concerning constrained optimization problems as well as some elements of complexity theory. We also present a taxonomy of scheduling problems and situate the scheduling problems studied in this thesis within this taxonomy. In Chapter 2 we present and illustrate some of the main methods used in the literature to solve these problems.

In the second part of the thesis we introduce and study a new constraint for scheduling problems (named *group cumulative constraint (GC)*), which we use to model the previously mentioned problem of available space to process several orders in parallel. In Chapter 3 we formally introduce this constraint and we study its links with related problems in the literature. We also give the proof of *NP*-completeness mentioned above and we study its impact on existing resolution methods. In particular, in Chapter 4 we propose different approaches to adapt these solving methods so that they can take into account this new constraint.

In the last part of this thesis, we experimentally evaluate the different methods mentioned above on the dataset provided by Infologic. In particular, in Chapter 5 we describe our dataset and our industrial context. We also mention several articles evaluating the interest of implementing scheduling techniques within the industry. In Chapter 6 we give the results of the experimental evaluation of the methods used on our dataset. In particular, we evaluate the impact of the different constraints used in the different scheduling problems on the methods presented. In Chapter 7 we evaluate the interest of using the method of *automatic algorithm selection*. This method uses *machine learning* techniques to automatically choose which algorithm to use according to the features of the instance to be solved. Finally, in Chapter 8 we study a certain scheduling problem by taking into account its dynamic aspect. In particular, the different commands to be performed are not all known in advance and are revealed while the algorithm is running. After a short presentation of the techniques used in the literature to deal with dynamic problems, we evaluate the impact of prohibiting the modification of future tasks to be performed by operators.

Publications

The constraint GC was introduced in a paper published at the *The Genetic and Evolutionary Computation Conference (GECCO)*[GNS20b] in 2020, with the automatic selection of parameters for *ant colony algorithms* (part of Chapter 7). The theoretical study of the constraint GC (including the proof of $\mathcal{NP}-$Completeness), as well as the algorithm combining CPO and ACO were published at the *international conference on principles and practice of CP* in 2020 [GNS20a].
We plan to publish in a journal article, the experimental results of Chapter 6 which evaluate the impact of the different constraints on the different algorithms. We also plan to include the adaptations considering the GC constraint described in Chapter 4, as well as the use of the *automatic algorithm selection* on the complete dataset (Chapter 7).

Notations

Input Data	
$\mathcal{J}$	set of jobs
n	number of jobs
j or $k \in \mathcal{J}$	lowercase letters used to designate jobs
p_j	processing time of job j
r_j	release date of job j
d_j	due date of job j
s_{jk}	sequence-dependent setup-time between jobs j and k
$\mathcal{M}$	set of machines
m	number of machines
$i \in \mathcal{M}$	lowercase letter used to designate machines
s_i	speed of machine i
$\mathcal{A}_i = ([b^i_1, e^i_1], [b^i_2, e^i_2], \ldots, [b^i_{\lvert\mathcal{A}_i\rvert}, e^i_{\lvert\mathcal{A}_i\rvert}])$	Available periods of machine i, with $b^i_1 < e^i_1 < b^i_2 < e^i_2 < \ldots < b^i_{\lvert A_i\rvert} < e^i_{\lvert A_i\rvert}$
$\mathcal{P}$	Set of groups, *i.e.* partition of $\mathcal{J}$ in $\lvert\mathcal{P}\rvert$ groups each job $j \in \mathcal{J}$ belongs to exactly one group $g \in \mathcal{P}$
g	lowercase letter used to designate groups in $\mathcal{P}$
h	time horizon
$\mathcal{H} = \{0, 1, \ldots, h\}$	all considered time points
$t \in \mathcal{H}$	lowercase letter used to designate time points

Given a set $\mathcal{X}$, we denote $\lvert\mathcal{X}\rvert$ its cardinality and $\mathscr{P}(\mathcal{X})$ its power set, i.e. the set of all subsets of $\mathcal{X}$ including the empty set and $\mathcal{X}$ itself (in the literature $\mathscr{P}(\mathcal{X})$ is sometimes also denoted $2^{\mathcal{X}}$).
Notation $\exists! x \in \mathcal{X}$ means that there exists a unique element x in $\mathcal{X}$.
$[l, u]$ denotes the set of all integers ranging from l to u. All the notations used are summarized in Fig 1.

Variables	
B_j	start time (or begin time) of job j
C_j	completion time (or end time) of job j
I_j	machines to which j is assigned
C_i	end time of machine i
	$C_i = \max_{j \in \mathcal{J}: I_j = i} C_j$
B_g	start time of group g
	$B_g = \min_{j \in g} B_j$
C_g	end time of group g
	$C_g = \max_{j \in g} C_j$
Performance measures	
$\mathcal{A}$	set of algorithms
$a \in \mathcal{A}$	lowercase letter used to designate algorithms
$\mathcal{I}$	set of instances
$x \in \mathcal{I}$	lowercase letter used to designate an instance
x_a^t	the value of the best solution found by a for x within t seconds
$x^* = \min_{a \in \mathcal{A}} x_a^{3600}$	the reference solution, the best solution found by any algorithm after one hour of computation
$ir_{a,x}^t = \begin{cases} 1 & \text{if } x_a^t = x^* \\ x^*/x_a^t & \text{otherwise} \end{cases}$	the inverse ratio
$t_{a,x}^{ref} = \begin{cases} \infty & \text{if } ir_{a,x}^{3600} < 1 \\ \underset{t \in [0,3600]}{\text{argmin}}\, ir_{a,x}^t = 1 & \text{otherwise} \end{cases}$	the time needed by algorithm a to find the reference solution on instance x
$t_{a,x}^{opt}$	the time needed by algorithm a to solve instance x and prove optimality
$mem_{a,x}$	the maximum amount of memory used

Figure 1: Used notations for scheduling problems

Part I

Background on Scheduling

Chapter 1

Definitions

Contents

1.1 Constrained Optimization Problems **21**
1.2 Scheduling problems **23**
1.2.1 Machine environment α 24
1.2.2 Constraints and job characteristics : field β 27
1.2.3 Objective functions : field γ 28
1.2.4 RCPSP 28
1.2.5 Thesis scope 29
1.3 Computational Complexity **32**
1.3.1 Decision problems 33
1.3.2 Complexity classes 33
1.3.3 Problem reduction 34
1.4 Discussion **34**

Constrained Optimization Problems (COPs) allow modeling real-life problems, in which a function must be optimized and some constraints must be satisfied. It gives a framework which allows us to study both solving methods and computational complexity of problems. In section 1.1 we introduce COPs. Among problems which can be modeled as COPs, lie the scheduling problems. Scheduling problems aim at assigning start and end times and resources to jobs. The literature regarding scheduling problems is vast, and their industrial applications are numerous. In section 1.2, we describe more precisely scheduling problems in order to define the scope of this thesis. Many of these problems are challenging from a computational point of view. In section 1.3, we introduce some background on computational complexity in order to better understand the challenge.

1.1 Constrained Optimization Problems

Many problems involve finding a set of values that satisfy some constraints. These problems are called Constraint Satisfaction Problems. In this section, we recall the main definitions related to these problems. We refer the reader to [RVBW06] for more details.

A *Constraint Satisfaction Problem* (CSP) involves finding solutions to a *constraint network*, that is, assigning values to variables so that constraints are satisfied. Each variable has a domain, which is the set of values that may be assigned to it. In this thesis, we only consider finite domains and, without loss of generality, we assume that all domains only contain integer values, *i.e.*, are finite subsets of $\mathbb{Z}$.

More formally, let us first define what is a constraint.

Definition 1.1.1 (Constraint)**.** A constraint c is a relation defined on a sequence of variables $X(c) = (x_{i_1}, \ldots, x_{i_{|X(c)|}})$, called the *scope* of c. c is the subset of $\mathbb{Z}^{|X(c)|}$ that contains the combinations of values $\tau \in \mathbb{Z}^{|X(c)|}$ that satisfy c. $|X(c)|$ is called the *arity* of c.

A constraint may be defined in intention, by using mathematical operators, or in extension, by listing all the tuples in the relation.

Example 1.1.1. The constraint $c \equiv x_1 < x_2 \wedge x_2 < x_3$ is the relation defined in intention that contains every tuple $(a, b, c) \in \mathbb{Z}^3$ such that $a < b$ and $b < c$. The tuple $(4, 7, 8)$ satisfies c whereas the tuple $(4, 1, 2)$ does not satisfy c. The arity of c is 3.

The constraint $c' \equiv \{(1, 2, 3), (2, 1, 3)\}$ is the relation defined in extension which is satisfied by two tuples: (1,2,3) and (2,1,3).

Definition 1.1.2 (CSP)**.**
A CSP is defined by a triple (X, D, C) such that:

- $X = (x_1, \ldots, x_n)$ is a finite sequence of integer variables;
- $D = D(x_1) \times \ldots \times D(x_n)$ is the domain for X, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values that may be assigned to the variable x_i;
- $C = \{c_1, \ldots, c_m\}$ is a set of constraints such that, for each constraint $c_i \in C$, every variable in $X(c_i)$ belongs to X.

The variables of a CSP and the scheme of a constraint c_i are sequences of variables and not sets because the order of values matters for tuples in D or c_i. However, we may use set operators on sequences. In particular, given two constraints c_i and c_j, we denote $X(c_i) \subseteq X(c_j)$ the fact that every variable in the scope of c_i also belongs to the scope of c_j, whatever their order in the scopes. Also, given a constraint c and a variable x, we denote $x \in X(c)$ the fact that x belongs to the scope of c.

Given a tuple τ on a sequence of variables Y, and another sequence of variables $W \subset Y$, we denote $\tau[W]$ the restriction of τ for the variables of W, ordered according to W.

Solving a CSP involves assigning variables to values so that constraints are satisfied.

Definition 1.1.3 (Assignment)**.**
Let (X, D, C) be a CSP.

- An *assignment* A on $Y = (x_1, \ldots, x_k) \subseteq X$ is a tuple of values $(v_1, \ldots, v_k)$ such that v_i is the value assigned to x_i. $A[x_i]$ denotes the value assigned to x_i in A, *i.e.*, v_i.
- An assignment A on Y is *valid* if for all $x_i \in Y, A[x_i] \in D(x_i)$.
- An assignment A on Y is *partial* if $Y \subset X$ and *complete* if $Y = X$.
- An assignment A on Y is *locally consistent* if it is valid and for every $c_i \in C$ such that $X(c_i) \subseteq Y$, $A[X(c_i)]$ satisfies c_i. If A is not locally consistent, it is *locally inconsistent.*
- A *solution* is a complete assignment A on X which is locally consistent. The set of solutions of (X, D, C) is denoted $sol(X, D, C)$.
- An assignment A on Y is *globally consistent* (or consistent) if it can be extended to a solution (*i.e.*, there exists $A' \in sol(X, D, C)$ with $A = A'[Y]$).

Example 1.1.2. Let us consider the following CSP:

- $X = \{x_1, x_2, x_3\}$;
- $D(x_1) = \{1, 2, 3, 4\}, D(x_2) = \mathbb{N}, D(x_3) = \{3, 4, 5, 6\}$;

- $C = \{c_1, c_2, c_3\}$ where
  - $c_1 \equiv x_1 \geq x_3$
  - $c_2 \equiv x_2 \geq x_1 + x_3$
  - $c_3 \equiv 2 * x_1 \leq x_2$

The complete *assignment* $A = (4, 9, 3)$ on $Y = (x_1, x_2, x_3)$ is a solution for this CSP.
For this CSP, the partial and valid assignment $A = (2, 4)$ on $Y = (x_1, x_2)$ assigns 2 to $x_1$and 4 to x_2. A is locally consistent because it satisfies the constraint c_3, which is the only one such that $X(c_3) \subseteq Y$ (indeed x_3 is both in $X(c_1)$ and in $X(c_2)$). However, A is not globally consistent because it cannot be extended to a solution. Indeed whatever the value of x_3 in D_{x_3}, c_1 cannot be satisfied.

Definition 1.1.4 (Constrained Optimization Problem (COP))**.**
A COP is defined by a quadruplet (X, D, C, f) such that (X, D, C) is a CSP, and $f : X \to \mathbb{R}$ is an objective function which is to be optimized.

Example 1.1.3. Let us consider the CSP defined in example 1.1.2, where we add the following objective function: $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$, which we try to minimize.
The solution $(3, 6, 3)$ minimizes f.

Many real-life problems can be modeled as COP. In this thesis, we especially focus on scheduling problems described more precisely in the next section.

1.2 Scheduling problems

The term *scheduling* is used to describe the activities which try to set order and time for planned events. Depending on the context, the events are sometimes designated by the terms *jobs*, *operations* or *activities*. Moreover, along with order and time, one sometimes wants to assign resources while scheduling events.

Scheduling problems are age-old problems. The idea of splitting projects into smaller activities and sequencing those activities exists since complex projects exist. Projects like pyramids in the Antiquity certainly needed scheduling in some way. Since that time, the way problems are modeled as scheduling ones and the methods used to schedule tasks have considerably evolved. The reader interested in the history of scheduling should refer to [HW06]. The evolution of scheduling as computer science is very close to computer science evolution itself. It means that with the development of computers in the mid-twentieth century, scheduling problems were among the first problems solved using computers. Already at that time, scheduling problems interested both academics and industries. The work of George Dantzig [Cot06] illustrates this interest as a theoretical point of view, whereas the CPM and PERT methods (1957), developed respectively by the company *E. I. du Pont de Nemours and Company* and by the U.S. Navy Special Projects Office, Bureau of Ordnance (SPO), illustrate scheduling theories' applications.

Nowadays, there exist many variants for scheduling problems. [Pin16] describes a wide range of them. This section aims at describing some of these variants.

At the core of the scheduling theory lies the notion of *jobs*. We denote $\mathcal{J}$ the set of all the jobs of the considered problem. A job has a set of characteristics that depends on the problem at hand. For example, in simplest models, a job j only has a processing time p_j, whereas in more complex models j may also have a release date r_j (*i.e.*, the earliest time at which j can be processed), a due date d_j (*i.e.*, the time at which the processing of j should be finished), ...
Along with the jobs, scheduling models often use a set $\mathcal{M}$ of machines. We denote m the cardinality of $\mathcal{M}$ (*i.e.*, $m = |\mathcal{M}|$), and we use i to designate machines. Machines also have characteristics that depend on the problem at hand.

Machines are not essential to define scheduling problems. There is a complete branch of scheduling theory which does not use (or not systematically) machines: in that case, we speak about Project Scheduling, or more precisely, about Resource-Constrained Project Scheduling Problem (RCPSP).

Symbol	Definition
	α
1	single machine model
P	identical parallel machines
Q	uniform parallel machines
R	unrelated parallel machines
F	Flow Shop Problem
J	Job Shop Problem
O	Open Shop Problem
PS	(Resource-Constrained) Project Scheduling Problem
	β
r_j	presence of release dates
M_j	machines eligibility
$brkdwn$	machines scheduled breakdowns
$prmp$	preemption authorized
$prec$	precedence constraints
s_{jk}	sequence-dependent setup times
$fmls$	job families
	γ
C_{max}	Minimizing the *makespan*
C_j	Minimizing the sum of the completion times
T_j	Minimizing the sum of the jobs' *tardiness*
L_{max}	Minimizing the maximum jobs' *lateness*
E_j	Minimizing the sum of the jobs' *earliness*
$E_j + T_j$	Minimizing the sum of the jobs' earliness and the sum of the jobs' tardiness
U_j	Minimizing the number of late jobs
$-$	No objective function, just find a schedule that respects all the constraints

Figure 1.1: Some possible values for α, β, and γ in the Graham notation

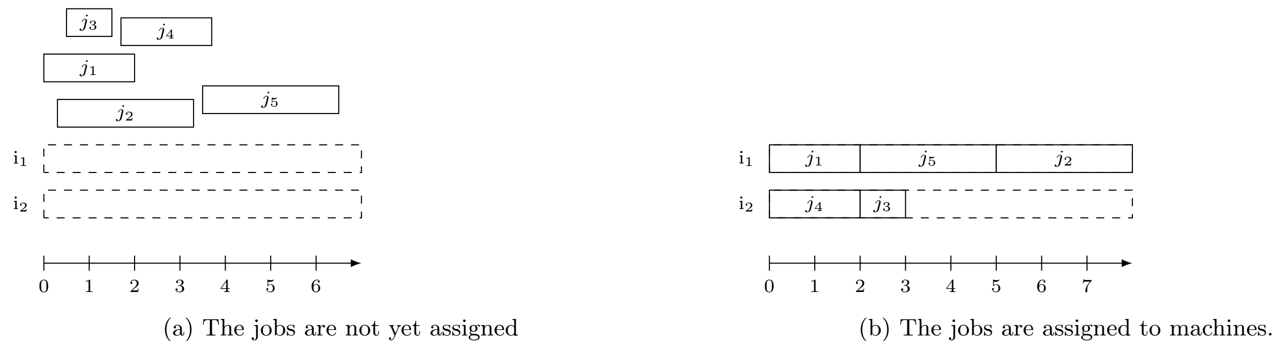
As aforementioned, there are many scheduling problems, and taxonomy has emerged to classify scheduling problems [GLLK79]. Hence scheduling problems are often described using the notation $\alpha|\beta|\gamma$ (called the Graham notation) where α describes the machine environment, β describes the characteristics of the jobs and constraints of the problem, and γ describes the objective function. Fig 1.1 summarizes the different values α, β, and γ can take and the next sections describe more precisely these values.

1.2.1 Machine environment α

α can take values into two subsets: on the first hand, there are the values that describe an environment with parallel machines (values **1**, P, Q and R), and on the other hand, there are the values which describe an environment with machines in series (F, J and O).

Single Machine Models $\alpha = 1$

In single machine models, one aims at scheduling a set of jobs on a single machine. Despite their simple formulation, single machine models are exciting. Scheduling problems on more than one machine are often solved using decomposition, where subproblems consist in scheduling on a single machine. Furthermore, single machine problems have interesting



(a) The jobs are not yet assigned

(b) The jobs are assigned to machines.

Figure 1.2: An example of a problem where $\alpha = P$ with two identical parallel machines. The length of a job represents its duration.

properties to calculate bounds or approximation schemes, which are in general much more complicated in other scheduling environments. It is also essential to highlight single machine models' pedagogic interest: it is often easier to find or explain theories in single machine scheduling problems than in a more complex environment. Moreover, it is not rare to consider a property in a multi-machine environment as a generalization of a single machine model's property.
Example of single machine models can be found in [AA14, LBG91, FMM01, NSBL18].

Parallel machine models

In parallel machine models, the set $\mathcal{M}$ consists of several machines that can process the jobs. Any of the machines can process the jobs in $\mathcal{J}$, and each job needs to be processed on only one machine. Three values for α model such environment :

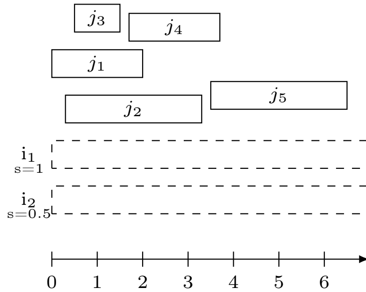
- *Identical parallel machines* $\alpha = P$: in that case, the machines in parallel are identical, which means that the duration of a job on any machine is equal to its processing time. Fig 1.2 shows an example of such an environment. The length of a job represents its duration. [WWY+18, Mok04, CGT95, MSCK09] study parallel machine scheduling problems. It is important to notice that even if the machine environment is the same, the problem can differ according to the constraints and the objective functions.
- *Uniform parallel machines* $\alpha = Q$: in that case each machine has its own speed. It means that if a job j with processing time p_j is scheduled on a machine i with speed s_i then the duration of j is equal to $\frac{p_j}{s_i}$. Fig 1.3 shows an example of such environment. [FL13] gives an example of such machine environment.
- *Unrelated parallel machines* $\alpha = R$: in that case the duration of the job j on machine i is equal to p_{ij} and is independent of the duration $p_{i'j}$ of the job j on another machine i'. Fig 1.4 shows an example of such environment. [PFG04] presents a survey about unrelated parallel machines scheduling with different objective functions, [FR12] also presents a problem with such environment.

P, Q, and R are sometimes followed by indices m to indicate the number of machines in parallel. For example, P_3 indicates that there are three machines in parallel.

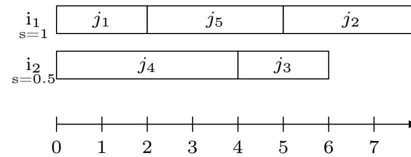
Machine in series - Multi-stage models - Shop problems

We use different names to designate the cases where machines are in series : *multi-stage models* or *shop problems*.
Three main variants exist for these models: the *Flow Shop Problem*, the *Job Shop Problem* and the *Open Shop Problem*.

In the *Flow Shop Problem* [RTF18] (denoted with $\alpha = F$ in the Graham notation), with m machines, each job j consists of m operations $o_{j,1}, o_{j,2}, \ldots, o_{j,m}$ where $o_{j,k}$ must be scheduled on machine k for all $k \in \mathcal{M}$. Hence, the route followed

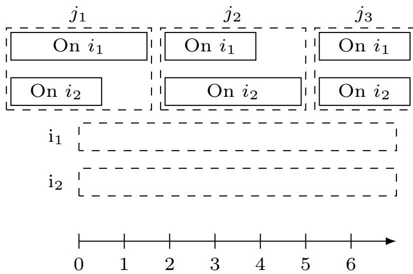


(a) The jobs are not yet assigned. Machine i_2 has a speed of 0.5

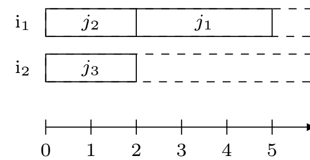


(b) The jobs are assigned to machines. Jobs assigned to i_2 takes twice more time to complete than if they were scheduled on i_1

Figure 1.3: An example of a problem where $\alpha = Q$ with two parallel machines where each machine has its own speed.



(a) There are three jobs. The job durations depend on the machine and are unrelated. For example j_1 is longer on i_1 than on i_2 whereas job j_2 is longer on i_2 than on i_1. Job j_3 has the same duration on both machines.



(b) Once assigned, job j_2 has duration 2 because it is on machine i_1, job j_1 has duration 3 and job i_3 has duration 2.

Figure 1.4: An example of a problem where $\alpha = R$ with two unrelated parallel machines. The job durations depend on the machine they are assigned to.

by the jobs is the same for every job. Each operation has its own duration on each machine: $p_{j,k}$ for the kth operation of job j.

In the *Job Shop Problem* (denoted with $\alpha = J$ in the Graham notation), each job j consists of n_j operations $o_{j,1}, \ldots, o_{j,n_j}$ where each operation must be executed on a specific machine and where operation $o_{j,i+1}$ cannot start before operation $o_{j,i}$ has ended.
The Job Shop problem is a well-studied one, and many articles deal with it. [MCZ10, CK16] give surveys about it with some references.

In the *Open Shop Problem* (denoted $\alpha = O$ in the Graham notation), each job j consists of n_j operations $o_{j,1}, \ldots, o_{j,n_j}$ where each operation must be executed on a specific machine but, contrary to the *job shop problem* the operation can be executed in any order. However, two operations of the same job cannot be simultaneously under execution. [HZC+19] gives a survey of several techniques used to solve open shop problems.

A natural extension of these problems involves combining parallel machine scheduling with multi-stage models. Such generalizations are known under the name *flexible* (for example *Flexible Flow Shop* (denoted FF or FF_m), *Flexible Job Shop* (denoted FJ or FJ_m) or *Flexible Open Shop* (denoted FO or FO_m)). For example, in the *Flexible Flow Shop*, there are several stages, and each stage consists of several parallel machines. Hence the jobs are sets of operations, whose first operation needs to be processed on one of the parallel machines of the first stage, the second operation needs to be processed on one of the parallel machines of the second stage, and so on.

The website [Des20] offers several instances of different scheduling problems (unrelated machines problems, flowshop problems, ...).

1.2.2 Constraints and job characteristics : field β

The following values can be used for the β field :

- r_j: to indicate that there are *release dates* on the jobs. A job j with a release date r_j cannot start its processing before time r_j.
- *brkdwn*: to model scheduled breakdowns on the machines (it models shifts in the teams or scheduled maintenance). [CHG19] presents mathematical models to deal with such constraints, and [LLSX17] presents a special case where scheduled maintenance operations have periodic recycles. [KH14] gives a survey of parallel machine scheduling under availability constraints.
- *prmp*: to indicate *preemptions*. A job is preemptable if it can be interrupted and resume later. The amount of processing a preempted job already has received is not lost. Hence when resumed, only the remaining processing time needs to be scheduled.
- *prec*: to indicate precedence constraints. It indicates that a job cannot start before another job is ended. There are models with other types of precedence constraints: for example, a job cannot start before another job is started.
- M_j: to indicate that some jobs can be scheduled only on a subset of machines in some models. [EO11] describes a model with such constraints.
- s_{jk}: to indicate *sequence dependent setup times*. If job k is scheduled just after job j on a same machine, then a setup time is required (of length s_{jk}) before starting the processing of k. [All15] give surveys of papers which considers sequence-dependent setup-times between jobs.
- $fmls$: to indicate *job families*. It is a generalization of the sequence-dependent setup times. In that case, the jobs are partitioned into families. There are no setup times between two jobs of the same family, but there exist setup times when one machine switches from one family to another.

The list is not exhaustive. There exist many other possible values to describe variants of scheduling problems. The interested reader may refer to [Pin16].

1.2.3 Objective functions : field γ

There exist many objective functions considered in the literature. In some of them, jobs have *due dates*. For a given job j, we denote d_j its due date. Furthermore, we denote C_j the completion time of a job (i.e., the time at which a job ends in a given schedule). The main considered objective functions are :

- C_{max} : Minimizing the *makespan*, i.e. $min \max_{j \in J} C_j$. It corresponds to minimizing the time at which the last executed job ends.
- C_j : Minimizing the sum of the completion time, i.e. $min \sum_{j \in J} C_j$.
- T_j : Minimizing the sum of the *tardiness* of the jobs, i.e. $min \sum_{j \in J} T_j$, where, for a job j, T_j is defined as $T_j = max(0, C_j - d_j)$.
- L_{max} : Minimizing the maximum *lateness* of the jobs, i.e. $min \max_{j \in J} C_j - d_j$.
- E_j : Minimizing the sum of the *earliness* of the jobs, i.e. $min \sum_{j \in J} max(0, d_j - C_j)$.
- $E_j + T_j$: Minimizing the sum of the earliness of the jobs and the sum of the tardiness of the jobs, i.e. $min \sum_{j \in J} |d_j - C_j|$ where $|.|$ denotes the absolute value. In that case, we want that the jobs complete as close as possible to their due dates.
- U_j : Minimizing the number of late jobs, *i.e.*, if we denote U_j (called *unit penalty function*) the value such that $U_j = \begin{cases} 1 & \text{if } C_j > d_j \\ 0 & \text{otherwise} \end{cases}$, then the objective function is $min \sum_{j \in J} U_j$. In the literature, the meaning of U_j is sometimes inverted (meaning that U_j equals one if job j is not late). In that case, the objective is to maximize the sum of the U_j; and the value of the sum is called the *throughput*.
- -: in that case, there is no objective function, and the goal is to find a schedule that satisfies all the constraints of the problem.

It is worth mentioning that when minimizing a sum, we can assign a weight to the different jobs. For example, instead of minimizing $\sum T_j$, it is often the case that the objective is to minimize $\sum w_j T_j$ where w_j is the weight associated with job j. It models the case where some jobs are more important than others. Furthermore, when we consider two objectives, we can have two distinct weights for the jobs. For example some try to minimize $\sum w'_j E_j + w''_j T_j$. It gives more weight to tardiness than earliness (or vice versa).

Fig 1.5 shows the value of the different objective functions according to the completion time of a job.

1.2.4 Resource-Constrained Project Scheduling Problem (RCPSP)

Project Scheduling (PS) is a variant of scheduling problems in which there are no machines. To be more precise, machines are not mandatory in Project Scheduling Problems. However, machines can still be modeled in this framework as a specific resource, and hence Project Scheduling can be seen as a generalization of Machine Scheduling Problems (MS). In Project Scheduling, precedence constraints are (almost) always considered. Furthermore, in Resource-Constrained Project Scheduling Problem (RCPSP) lies the notion of *resources*. The jobs consume the resources, and the quantity of available resources is limited. Some resources are limited on a per-instant basis (we speak about renewable resources), some others are limited for the whole project (non-renewable resources), and some others are limited on both a per-instant basis and for the whole project (doubly constrained resources). There also exist variants which consider the minimization of resource consumption as an objective (instead of a constraint).

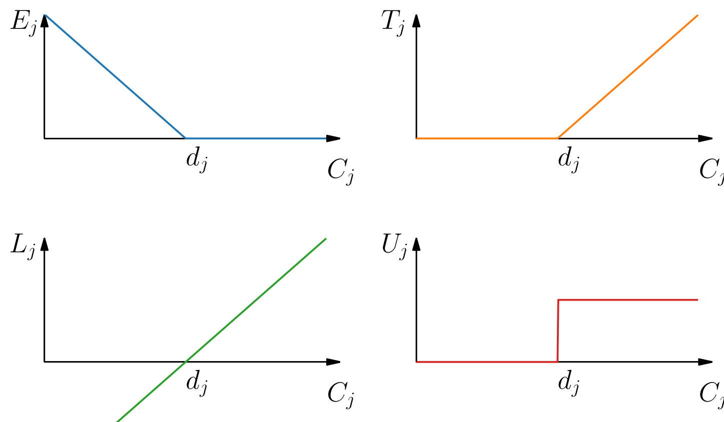


Figure 1.5: Evolution of the value of the objective function with respect to the completion time C_j of one job j when $\gamma = E_j$ (top left), T_j (top rigt), L_j (bottom left) and U_j (bottom right).

The interested reader can refer to the surveys [BDM+99] and [HB10] or to the book [ADN08] for further details on RCPSP.

1.2.5 Thesis scope

The previous subsections describe a general scope for scheduling problems. It allows us to situate the context of this thesis. The applicative context of this thesis is described in Chapter 5. In this context, we only consider variants of the parallel machine scheduling problems.
For each of these problems, we give a mathematical model (in the COP formalism) and we describe classical lower bounds for the objective function. These lower bounds are essential to prove optimality of solutions. For example, if an algorithm finds a solution whose value is equal to the lower bound, we know that this solution is optimal, and we can stop the search.

Assignment problem $P||C_{max}$

This problem is not really a scheduling problem in the sense that the start and end time of each job do not matter. It consists in minimizing the makespan while assigning jobs to identical parallel machines. Start and end times of jobs do not matter because we are only interested in the time at which each machine ends, and for a given machine, that time can easily be computed by summing all the durations of the jobs assigned to it. Hence we are only interested in the machine to which each job is assigned. This problem represents the situation where a set of jobs must be divided over identical machines, and we are only interested in balancing the load between the different machines.

We can use the following COP to model this problem:

- **Input Data:** A set $\mathcal{J}$ of n jobs, such that each job $j \in \mathcal{J}$ has a processing time $p_j \in \mathbb{N}$, and a set $\mathcal{M}$ of m machines
- **Variables:** For each job j, a variable I_j which represents the machine that processes j. The domains of these variables are $\mathcal{D}(I_j) = \mathcal{M}$.
- **Constraints:** No constraint.
- **Objective function:** Minimizing the makespan, *i.e.*, $\min \max_{i \in \mathcal{M}} \sum_{j \in \mathcal{J} : I_j = i} p_j$

In this assignment problem, a lower bound can be computed using the following formula:

$$lb = \left\lceil \frac{\sum_{j \in J} p_j}{m} \right\rceil$$

where m is the number of machines, and $\lceil x \rceil$ is the least integer greater than or equal to x. It corresponds to a perfect load where each machine has the same amount of work.
It is worth mentioning that when there is only one machine, the problem is trivial because all jobs are assigned to the unique machine.

Release date and due date $P|r_j|\sum T_j$

It is a basic parallel machine scheduling problem where we try to minimize the sum of tardiness while taking into account the jobs' release dates. This problem is a scheduling one because in order to compute the value of the objective function for one solution, we must know the start and end times of each job.

We can use the following COP to model this problem:

- **Input Data:** A set $\mathcal{J}$ of n jobs, such that each job $j \in \mathcal{J}$ has a processing time $p_j \in \mathbb{N}$, a release date r_j and a due date $d_j \in \mathbb{N}$, a set $\mathcal{M}$ of m machines, and a time horizon $h \in \mathbb{N}$
- **Variables:** For each job j, variables I_j and B_j represent the machine that processes j and the start time of j, respectively. The domains of these variables are $\mathcal{D}(I_j) = \mathcal{M}$ and $\mathcal{D}(B_j) = [r_j, h]$.
- **Constraints:** Jobs assigned to a same machine do not overlap, *i.e.*,

$$\forall \{j_1, j_2\} \subseteq J, I_{j_1} \neq I_{j_2} \vee [B_{j_1}, B_{j_1} + p_{j_1}] \cap [B_{j_2}, B_{j_2} + p_{j_2}] = \emptyset \tag{1.1}$$

- **Objective function:** Minimizing

$$\sum_{j \in \mathcal{J}} T_j \tag{1.2}$$

  where for each job j, T_j is defined as
  $T_j = \max(0, C_j - d_j)$ and C_j is defined as $C_j = B_j + p_j$.

In order to model release dates, one can use the constraint $\forall j \in \mathcal{J}\ B_j \geq r_j$; however, this is simpler to model it by restricting the domain of B_j to $[r_j, h]$ as done here.

In this basic scheduling problem, a lower bound can be computed using the following formula:

$$lb = \sum_{j \in J} \min_T(j)$$

where $\min_T(j)$ is the minimum tardiness of the job j which is equal in that case to $\min_T(j) = \max(0, r_j + p_j - d_j)$. It is often the case that $\min_T(j)$ is equal to zero, but there can exist jobs which have their release dates so close to their due dates that they are necessarily late (even in an optimal schedule).

Speed problem $Q|r_j|\sum T_j$

In this problem, we consider the machines' different speeds case. There are still release dates, and the objective function is also to minimize the total tardiness. Hence this problem is very close to the previous one apart from the machines' speeds.

To model this problem, we can use the same input data as for the previous case, except that we now have a speed s_i for each machine i. The variables are also identical, and the release date constraint is the same. The constraint 1.1 is modified in order to take machines' speed into account:

$$\forall \{j_1, j_2\} \subseteq J, I_{j_1} \neq I_{j_2} \vee \left[B_{j_1}, B_{j_1} + \frac{p_{j_1}}{s_{I_{j_1}}}\right] \cap \left[B_{j_2}, B_{j_2} + \frac{p_{j_2}}{s_{I_{j_2}}}\right] = \emptyset \tag{1.3}$$

The objective function also consists in minimizing 1.2, with the tardiness of a job defined as $T_j = \max(0, C_j - d_j)$, but this time $C_j = B_j + \dfrac{p_j}{s_{I_j}}$

In this uniform parallel scheduling problem, a lower bound can be computed using the same formula as with the identical parallel machine scheduling problem:

$$lb = \sum_{j \in J} \min_T(j)$$

Nevertheless, here $\min_T(j)$ the minimum tardiness of the job j must consider that not all the machines have the same speed. More precisely, we assume that all the jobs can be scheduled on the fastest machine to calculate the lower bound. Hence we have

$$\min_T(j) = \max\left(0, r_j + \frac{p_j}{s^*} - d_j\right)$$

where s^* is the speed of the fastest machine, *i.e.* $s^* = \max_{i \in M} s_i$

Sequence dependent setup times problem $Q|r_j, s_{jk}|\sum T_j$

In this problem, we consider an additional constraint, which is the presence of sequence-dependent setup-times. If job k is scheduled just after job j on the same machine, then a setup time (of length s_{jk}) is required before starting the processing of k.

To model this problem, we can adapt the COP given for the previous problem. We have an additional input data, which is the sequence-dependent setup-times s_{j_1,j_2} between each ordered pair of jobs $(j_1, j_2) \in J^2$ with $j_1 \neq j_2$. In our context, the setup-times are symmetric, meaning that $s_{j_1,j_2} = s_{j_2,j_1}$ for every jobs $\{j_1, j_2\} \subseteq J$. It is also important to notice that there is no setup-times for the first job of a machine.
The constraint 1.1 is replaced by:

$$\forall i \in \mathcal{M}, \forall j \in \{j' \in \mathcal{J} : I_{j'} = i \text{ and } succ(j) \neq \emptyset\}, B_k >= \max(B_j + p_j/s_i, r_k) + s_{j,k} \tag{1.4}$$

where $succ(j) = \{j' \in \mathcal{J} : I_{j'} = I_j \text{ and } B_{j'} > B_j\}$ is the set of all jobs that are assigned on i and start after j, and $k = \underset{k' \in succ(j)}{argmin}\ B_{k'}$ is the job of $succ(j, i)$ that has the smallest start time.

For this case, the lower bound is the same as in the case with just speeds on the machine. It means that we consider that there are no setup times for the lower bound.

Breaks problem $Q|r_j, brkdwn|\sum T_j$

In this case there are scheduled breakdowns on the (uniform parallel) machines (we do not consider the setup times anymore here). It is essential to notice that in this case, a job can be interrupted by a break and resume after it without any loss on the work already done. It means that if a job needs x units of time to be computed on a given machine and $y < x$ units have already been completed on the machine before the break, then only $x - y$ remain to compute after the break. We use it to represent the workers' breaks in our application context.

We can adapt the COP given for $Q|r_j|\sum T_j$ problem to model this problem. All the input data are reused, but, for this problem, we have additional data: for each machine $i \in \mathcal{M}$ we have a set of periods $\mathcal{A}_i \subseteq \mathcal{H}$ during which the machine

is available. This set can be represented as

$$\mathcal{A}_i = ([b^i_1, e^i_1], [b^i_2, e^i_2], \ldots, [b^i_{N_{\mathcal{A}_i}}, e^i_{N_{\mathcal{A}_i}}])$$

where $N_{\mathcal{A}_i}$ is the number of intervals in the list, $[b^i_l, e^i_l]$ for $l \in [1, N_{\mathcal{A}_i}]$ is the lth period of availability of machine i, and the bounds on the available periods are such that $b^i_1 < e^i_1 < b^i_2 < e^i_2 < \ldots < b^i_{N_{\mathcal{A}_i}} < e^i_{N_{\mathcal{A}_i}}$.
Once again, we need to replace the constraint 1.1 by:

$$\forall i \in \mathcal{M}, \forall j \in \mathcal{J}, I_j = i, \forall k \in succ(j), |[B_j, B_k] \cap \mathcal{A}_i| \geq p_j/s_i \quad (1.5)$$

where $|.|$ is the interval length.
The objective function also consists in minimizing 1.2 but this time

$$C_j = \min_{t \in [0,h]} \left(\left|[B_j, t] \cap \mathcal{A}_{I_j}\right| \geq \frac{p_j}{s_{I_j}} \right)$$

Once again, a lower bound can be computed in a similar way as for the speed problem $Q|r_j|\sum T_j$, meaning that we do not consider the breaks in the computation of the lower bound.

Breaks and sequence-dependent setup times problem $Q|r_j, brkdwn, s_{jk}|\sum T_j$

This last case is a combination of the two previous ones. It is important to notice that the setup-times considered in our problem are due to human activity (adjust a machine to the used product or change the used packaging roll). Thus, these setup-times cannot be executed during the breaks (as workers are not present during the breaks). Depending on the context, in the literature, setup-times can sometimes be executed during breaks (for example, if the setup time corresponds to the action of heating an oven which can be executed even if workers are not present).

The following COP model this problem:

- **Input Data:**
  - A set $\mathcal{J}$ of n jobs, such that each job $j \in \mathcal{J}$ has a processing time $p_j \in \mathbb{N}$, a due date $d_j \in \mathbb{N}$ and a release date $r_j \in \mathbb{N}$
  - Sequence-dependent setup-times $s_{j_1,j_2} \in \mathbb{N}$ between each pair of jobs $\{j_1, j_2\} \subseteq J$
  - A set $\mathcal{M}$ of m machines such that each machine $i \in \mathcal{M}$ has a speed s_i, and a set of periods $\mathcal{A}_i \subseteq \mathcal{H}$ during which the machine is available
- **Variables:** For each job j, variables I_j and B_j represent the machine that processes j and the start time of j, respectively. The domains of these variables are $\mathcal{D}(I_j) = \mathcal{M}$ and $\mathcal{D}(B_j) = [r_j, h]$.
- **Constraints:** Jobs do not overlap and sequence-dependent setup-times start after the jobs' release-date

$$\forall i \in \mathcal{M}, \forall j \in \mathcal{J}, I_j = i, \forall k \in succ(j), |[B_j, B_k] \cap \mathcal{A}_i| \geq p_j/s_i + s_{j,k} \wedge |[r_k, B_k] \cap \mathcal{A}_i| \geq s_{j,k} \quad (1.6)$$

- **Objective function:** Minimizing $\sum_{j \in \mathcal{J}} T_j$ with $T_j = \max(0, C_j - d_j)$ where

$$C_j = \min_{t \in [0,h]} \left(\left|[B_j, t] \cap \mathcal{A}_{I_j}\right| \geq \frac{p_j}{s_{I_j}} \right)$$

1.3 Computational Complexity

Most of the scheduling problems introduced in the previous section are computationally challenging. In this section, we first introduce the main complexity classes, to better understand the challenge, and then we introduce reduction procedures which are used to characterize the complexity classes of new problems. The interested reader can refer to [AB09] for further information about computational complexity.

1.3.1 Decision problems

Definition 1.3.1 (Decision problems)**.**
A *decision problem* is a problem that can be posed as a *yes-no question* of the input values.

Every CSP π (as defined in section 1.1) can be seen as a *decision problem*, where the *yes-no question* is: "Does there exist a *solution* to problem π ?".

It is important to highlight links between COP and CSP. Given a COP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, we can associate a CSP to it, which consists in the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ with an additional constraint $f(X) \leq h$, with h a number (which is a data of the CSP).

Example 1.3.1. Let us consider the scheduling problem which consists in minimizing the maximum lateness for a set of jobs on a single machine $1||L_{max}$. In the COP version of the problem, we are looking for a solution that minimizes L_{max}. In the CSP version, we are looking for a solution whose value for L_{max} is bounded by a given h.

This relation between a COP and a CSP is important. Let us assume that we have a procedure that solves the CSP; then, by dichotomy, we can find the optimal value of the COP by using this procedure.

Definition 1.3.2 (Instance of a problem)**.**
An *instance* of a problem is obtained by specifying the values of all its input parameters.

Example 1.3.2. If we consider the problem assignment problem $P||C_{max}$ defined in the section 1.2.5, an instance is obtained by specifying the number of jobs, the number of machines and the processing time of each job.
For example an instance of this problem consists in $\mathcal{J} = \{j_1, j_2, j_3\}$ with $p_{j_1} = 2$, $p_{j_2} = 1$, and $p_{j_3} = 2$, and $\mathcal{M} = \{i_1, i_2\}$.

Definition 1.3.3 (Size of an instance)**.**
The size of an instance is the number of *bits* needed to represent it. Given an instance I of a problem, we denote $|I|$ its size.

1.3.2 Complexity classes

An algorithm A is said to be *polynomial* if there exists a constant value k such that the number of elementary operations done by A to solve any instance of size n is in $O(n^k)$.

Definition 1.3.4 (Class $\mathcal{P}$ - polynomial)**.**
A decision problem π is said to be polynomial if there exists a polynomial algorithm that can solve any instance of π.
The class $\mathcal{P}$ (for polynomial) is the set of all the polynomial problems.

In a sense, the polynomial decision problems are easy to solve, meaning that the number of operations needed to solve them is not exponential with respect to the problem's size.

Example 1.3.3. Let us consider the $1||L_{max}$ problem. This problem can be solved in polynomial time using the Earliest Due Date rule (EDD, also known as Jackson's EDD rule, due to R. Jackson, who studied it in 1954). The EDD rule consists in sorting all the jobs according to their due date and assigning them to the machine in that order [Stu70].

Definition 1.3.5 (Class $\mathcal{NP}$ - nondeterministic polynomial)**.**
A decision problem π belongs to Class $\mathcal{NP}$ if there exists an algorithm that can solve any instance of π in polynomial time on a non deterministic Turing machine. We refer the reader to [AB09] for the definition of non deterministic Turing machines. A consequence of this definition is that a decision problem π belongs to Class $\mathcal{NP}$ if for each instance I of π such that the answer of I is yes, there exists a certificate $c(I)$ such that

- the size of $c(I)$ is polynomial with respect to the size of I
- the problem of deciding whether $c(I)$ is a correct solution is in $\mathcal{P}$.

Example 1.3.4. Most of the decision problems associated with scheduling problems belong to the $\mathcal{NP}$ class. For example let us consider the decision problem associated with the $P||C_{max}$ problem. A certificate is an assignment A for each

variable I_j associated with a job $j \in \mathcal{J}$. Deciding if A is a solution can be done in polynomial time by computing the makespan associated with A and checking that it is smaller than or equal to the given bound. Hence $P||C_{max} \in \mathcal{NP}$.

Definition 1.3.6 ($\mathcal{NP}-Complete$ problems)**.**
A decision problem π is $\mathcal{NP}-Complete$ if it is in $\mathcal{NP}$ and if it is the most difficult problem among all the problems in $\mathcal{NP}$. In other words, a decision problem π is $\mathcal{NP}-Complete$ if finding a polynomial algorithm to solve π implies that $\mathcal{P} = \mathcal{NP}$.

Example 1.3.5 ($P||C_{max}$)**.** The decision problem associated with $P||C_{max}$ is $\mathcal{NP}-Complete$ [Pin16].

Definition 1.3.7 (NP-Hard problems)**.**
A decision problem is $\mathcal{NP}-hard$ if it is at least as difficult as the hardest problem in $\mathcal{NP}$.

Classes $\mathcal{P}$ and $\mathcal{NP}$ are defined for decision problems, and not for optimization problems. However, when the decision problem associated with an optimization problem π is $\mathcal{NP}-complete$, we say that π is $\mathcal{NP}-hard$.

1.3.3 Problem reduction

Problem reduction is used to demonstrate that a problem is at least as difficult as another problem.

Definition 1.3.8 (Problem reduction)**.**
Given two decision problems π_1 and π_2, a reduction from π_1 to π_2 is a function ϕ which transforms every instance of π_1 into an instance of π_2 and which verifies that, for every instance I of π_1, the answer for I is yes if and only if the answer for the instance $\phi(I)$ of π_2 is yes.

Reductions are used to build a hierarchy of problems. Indeed, if there exists a reduction ϕ from a problem π_1 to another problem π_2, then the complexity class of π_1 is upper bounded by both the complexity class of π_2 and the complexity of the reduction ϕ. In particular, if ϕ has a polynomial time complexity, and if π_1 is known to be $\mathcal{NP}-complete$, then we can conclude that π_2 is $\mathcal{NP}-hard$. Indeed, we can solve any instance I_1 of π_1 by reducing it to an instance $I_2 = \phi(I_1)$ of π_2 and then solving I_2. Hence π_2 is at least as hard as π_1 and if we could solve π_2 in polynomial time, then we could also solve π_1 in polynomial time (and in this case the two classes $\mathcal{P}$ and $\mathcal{NP}$ would be equal). Finally, if π_2 also belongs to $\mathcal{NP}$, then we can conclude that it is $\mathcal{NP}-complete$.

The first problem to have been proved to be $\mathcal{NP}-Complete$ is the SAT-Problem. The interested reader can refer to [Coo71] to have details about this proof.

Example 1.3.6 (Scheduling problems)**.** As mentioned in example 1.3.5, the decision problem associated with $P||C_{max}$ is $\mathcal{NP}-Complete$. $P|r_j|T_j$ is also $\mathcal{NP}-Complete$ [Pin16]. All the other scheduling problems mentioned in section 1.2.5 are generalization of $P|r_j|T_j$. Hence they also are $\mathcal{NP}-Complete$ (the reduction from $P|r_j|T_j$ to any other scheduling problem $P|\beta|T_j$ is trivial).

1.4 Discussion

In this chapter we introduced the notion of *Constrained Optimization Problems (COPs)*. Among the problems, which can be modeled as COPs, are the scheduling problems. We described three main classes of scheduling problems: the ones with parallel machines, the ones with machines in series, and the resource-constrained project scheduling projects. In this thesis we focus on parallel machines scheduling problems and more precisely on *uniform parallel machine scheduling problems*. We consider several constraints and objectives: sequence-dependent setup-times, scheduled breakdowns, minimization of the makespan, minimization of the jobs' tardiness sum.
We also introduced the notion of computational complexity in this chapter, with a framework which allows to classify problems (using polynomial reduction). In particular, we have seen that all the scheduling problems described in Section 1.2.5 are $\mathcal{NP}-hard$ problems, *i.e.*, they cannot be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$.

In order to find good solutions to solve $\mathcal{NP}-Hard$ COPs, several methods have been developed. Some of these methods are presented in the next chapter.

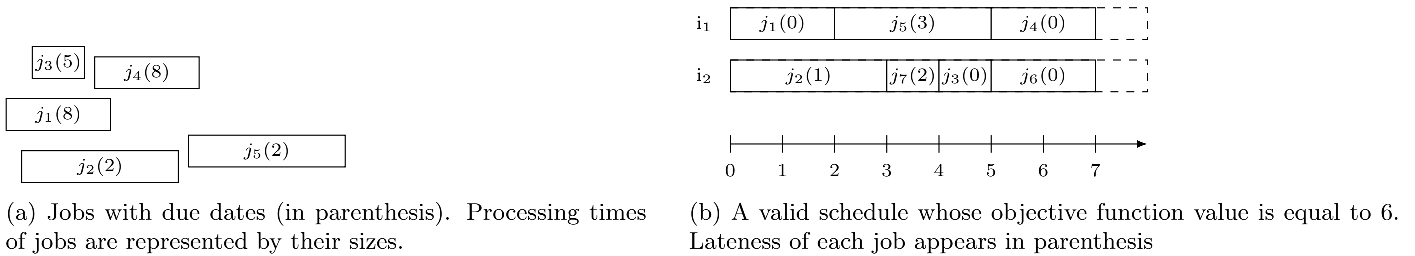
Chapter 2

Main solving approaches

Contents

2.1 Greedy Approaches **38**
2.1.1 Dispatching rules 38
2.1.2 Greedy Randomized Approaches 39
2.2 Ant Colony Optimization (ACO) **40**
2.3 Local Search **42**
2.3.1 Construction of an initial solution 42
2.3.2 Neighborhood 42
2.3.3 Choosing a neighbor in the neighborhood 44
2.4 Linear Programming **45**
2.4.1 Mixed Integer Linear Programming 47
2.4.2 Models for scheduling problems 47
2.5 Constraint Programming **50**
2.5.1 Type of data and constraint 50
2.5.2 Constraint propagation 51
2.5.3 Branch and propagate 51
2.5.4 Model and solvers 52
2.5.5 Models for scheduling problems 52
2.6 Discussion **56**

In this chapter we describe several methods which are used to solve COPs. We illustrate these methods on scheduling problems.
We first describe incomplete approaches, that quickly compute "good" solutions, without any guarantees on the quality of these solutions. In section 2.1 we present greedy approaches which aim at quickly finding good solutions of COPs. These greedy approaches are often problem-specific and highly depend on the problem's structure. Although there often exist greedy approaches for each COP, they can hardly be generalized to solve unrelated COPs. In section 2.2 we present *Ant Colony Optimization*, a meta-heuristic which greedily builds many solutions and which learns from previous constructions to improve next constructions. In section 2.3 we describe local search and especially tabu search, a method which improves a current solution by modifying it step by step. The idea of local search is to focus the search on solutions which are close to the incumbent solution. This notion of proximity between solutions is described more precisely in section 2.3.
Finally, we describe two exact approaches, that find optimal solutions (and prove their optimality), but have exponential time complexities in the worst case. In section 2.4 we present *Linear Programming*, the goal of which is to optimize a linear objective function, subject to linear equality and linear inequality constraints. Finally in section 2.5 we introduce *constraint programming* (CP). CP allows a user to define a problem in a declarative way, by means of variables and constraints, and then to solve it by using generic search procedures.



(a) Jobs with due dates (in parenthesis). Processing times of jobs are represented by their sizes.

(b) A valid schedule whose objective function value is equal to 6. Lateness of each job appears in parenthesis

Figure 2.1: A representation of an instance of the parallel machines scheduling problem with tardiness objective $P||\sum T_j$

Illustration and notations

In order to illustrate this chapter, we will use the scheduling problem $P|\beta|\sum T_j$ where $\beta \subseteq \{r_j, s_{jk}, brkdwn\}$ is the set of constraints formally described in section 1.2.5. This problem is known to be $\mathcal{NP} - Hard$ [Pin16]. Fig 2.1 shows an instance of this problem when $\beta = \emptyset$, and a solution whose value is 6.

We use several additional notations in this chapter. Given a scheduling problem, with a set of machines $\mathcal{M}$ and a set of jobs $\mathcal{J}$, and a (partial or complete) assignment A, we denote C_j the end of job j (or completion time). The expression of C_j depends on the considered problem. For example, when considering $P||\sum T_j$, we have $C_j = B_j + p_j$. However, with other constraints, the value of C_j can be a more complex expression. For example, if we consider the scheduling problem with scheduled breakdowns on machines, the end of a job is equal to its start time plus its duration and possibly the duration of all the breaks crossed by the job. We also denote C_i the end of machine $i \in \mathcal{M}$ in the assignment A, *i.e.*, $C_i = 0$ if no job has been assigned to i in A, and $C_i = \max\{C_j | j \in \mathcal{J}, I_j = i\}$ otherwise.

2.1 Greedy Approaches

Greedy approaches aim at quickly constructing a solution to a COP, which is as best as possible. Greedy approaches do not aim at proving the optimality of solutions. They are often used as a component of other methods. For example, when using local search, greedy approaches are used to generate a first solution which is then improved by local search (see section 2.3). Greedy approaches are also used to generate solutions in the ACO framework (see section 2.2).

2.1.1 Dispatching rules

When considering scheduling problems, dispatching rules are often used [Pin16]. The general procedure of dispatching rules is depicted in Algo. 1. The idea is to select the machine that ends the soonest, select a job among the unassigned ones, and assign that job to the selected machine. We repeat this procedure until all jobs are assigned.

The different dispatching rules vary in the way they select the next job to be assigned among the unassigned ones. Among dispatching rules, a subset of approaches is known as list scheduling heuristics. The idea of list scheduling is to sort all the jobs of $\mathcal{J}$ according to a given criterion and to assign them according to this order. Among the main used list scheduling heuristic, we can cite:

- *Earliest Due Date (EDD)*: jobs are sorted in increasing order of their due dates;
- *Longest Processing time first (LPT)*: jobs are sorted in decreasing order of their processing times;
- *Shortest Processing time first (SPT)*: jobs are sorted in increasing order of their processing times;

According to the problem features, some dispatching rules are more adapted than others. For example, SPT is optimal for $1|brkdwn|\sum C_j$ [Pin16]. LPT is well-suited for $P||C_{max}$. Indeed longest jobs are processed first, and then shortest

Algorithm 1: Greedy construction according to dispatching rules

Input : A parallel machine scheduling problem $P|\beta|\sum T_j$ with a set of jobs $\mathcal{J}$, and a set of machines $\mathcal{M}$
Output: A solution, i.e., a consistent assignment of all variables in $X = \{I_j, B_j, C_j | j \in \mathcal{J}\}$ (where I_j, B_j, and C_j represent the machine, start time, and completion time of job j, respectively.)

```
1 Cand ← J
2 while Cand ≠ ∅ do
3 |  i ← argmin_{i'∈M} C_{i'}
4 |  Select a job j in Cand
5 |  Cand ← Cand \ {j}
6 |  Assign i to I_j
7 |_ Assign to B_j and C_j the smallest possible values that satisfy constraints in β
```

jobs are assigned in order to balance the load between machines. Furthermore, for this problem we can compute the following bound [Pin16] :

$$\frac{C_{max}(LPT)}{C_{max}(OPT)} \leq \frac{4}{3} - \frac{1}{3m}$$

where $C_{max}(OPT)$ is the value of an optimal solution, and $C_{max}(LPT)$ is the value of the solution obtained following the LPT rule.
As already seen, *EDD* is optimal for $1||L_{max}$.

Another dispatching rule (not list-scheduling rule) is the *Minimum Slack* first rule *(MS)*. This rule selects at time t, when a machine is freed, among the remaining jobs the job j with the minimum slack: $\max(d_j - p_j - t, 0)$.

Among the main used greedy approaches, we can cite the *Apparent Tardiness Cost (ATC)* [VM87, Pin16]. This rule selects at time t, when a machine is freed, among the remaining jobs the job j which maximizes:

$$\frac{1}{p_j} \exp\left(-\frac{\max(0, d_j - p_j - t)}{K\overline{p}}\right)$$

where K is a parameter and $\overline{p}$ is the average processing time of the unscheduled jobs. The value of K can be set manually or calculated based on some features of the instance (number of jobs, mean processing time, number of machines, *due date tightness factor*, ...), more details can be found in [Pin16]. It is important to notice that if K is huge, then the rule is equivalent to the *SPT* rule. On the other hand, if K is very small, the rule is equivalent to the *Minimum Slack* rule for the jobs when there are no overdue jobs and to SPT for the overdue jobs if there are such jobs.

When considering sequence-dependent setup-times, the rule can be extended in order to take these setup-times into account. The idea is to find a compromise between the jobs' processing time, the jobs' slack, and the setup-times. This dispatching rule is called the *Apparent Tardiness Cost with setups (ATCS)* [LP97, Pin16] and consists in selecting at time t, when a machine i is freed, and the last job on i is k, among the remaining jobs the job j which maximizes:

$$R_j(t,k) = \frac{1}{p_j} \exp\left(-\frac{\max(0, d_j - p_j - t)}{K_1\overline{p}}\right) \exp\left(-\frac{s_{k,j}}{K_2\overline{s}}\right)$$

where K_1 and K_2 are parameters, $\overline{p}$ is the average processing time of the unscheduled jobs, and $\overline{s}$ is the average of the setup times of the jobs remaining to be scheduled.

2.1.2 Greedy Randomized Approaches

Greedy approaches build one solution. They have polynomial time complexities and (often) find good solutions in a small amount of time. However, when we have more time to spend on the search, we would like to use that time to find

better solutions. The first idea to do so is to introduce some randomness in the greedy approaches and to repeatedly build solutions until our time limit is reached (or another stopping criterion is met). This algorithm is called the *Greedy Randomized Search Procedure (GRS)* [FR95].

For example, to introduce randomness in the *ATCS* procedure, each time a job must be chosen among the unscheduled ones, instead of choosing the one that maximizes $R_j(t,k)$ we use a roulette wheel to select one job randomly. The probability are biased such that the probability of selecting job j is equal to $\frac{R_j(t,k)}{\sum_{j' \in Cand} R_{j'}(t,k)}$.

All the same, for the *EDD* rule, instead of selecting the job which maximizes $\frac{1}{d_j}$ we choose job j with probability $\frac{1/d_j}{\sum_{j' \in Cand} 1/d_{j'}}$

2.2 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) [CDM91, DS04] is a meta-heuristic which has been used to solve various optimization problems. It looks like *GRS*, but in order to improve the search, it learns from the already constructed schedules. The idea is to learn the components of the good solutions found so far to improve the next solutions' construction. The algorithm is depicted in Fig 2.

Many variants of *ACO* algorithms exist. Among the most used ones we can cite *Ant Colony System (ACS)* [DG97], *Max-Min Ant System (MMAS)* [SH98], or *P-ACO (population based ACO)*[GM02]. In this thesis, we consider MMAS which has been shown to obtain good results [DS04].

ACO algorithms use pheromone structures to learn good solution components. The two most widely considered pheromone structures for scheduling problems are

- *Job structure*, where a pheromone trail $\tau(j,j')$ is associated with every couple of jobs $(j,j') \in \mathcal{J}^2$ to learn the desirability of scheduling j' just after j on a same machine;
- *Position structure*, where a pheromone trail $\tau(j,i,n)$ is associated with each triple $(j,i,\,n) \in \mathcal{J} \times \mathcal{M} \times [1,|\mathcal{J}|]$ to learn the desirability of scheduling job j at position n on machine i.

Many different pheromone structures have been proposed for solving scheduling problems, and a review of 54 of these algorithms may be found in [TNGF13]. Among these 54 algorithms, *Jobs* structure are used in 38 papers, and *Position* structure are used in 17 papers (some using both).

In line 2 of Alg. 2 pheromone trails are initialized. In the *MMAS* strategy, all pheromone trails are initialized to τ_{max}, where τ_{max} is a parameter.

Then, at each iteration of the loop lines 13-20, n_{ants} solutions are constructed in a greedy randomised way, where n_{ants} is a parameter which is used to control exploration (the larger n_{ants}, the stronger the exploration). At each iteration of the greedy construction (lines 15-18), a machine i and a job j are chosen, and j is scheduled on i, until all jobs have been scheduled. The choice of i is made according to some heuristics, which depend on the scheduling problem (a classical way to do is to select the machine which ends the soonest like in the dispatching rules).
The choice of j is made in a randomized way, according to a probability $p(j)$ which depends on two factors.

The heuristic factor $\eta(j)$ evaluates the interest of scheduling j on i and its exact definition depends on the scheduling problem. The heuristic factor is often chosen among the previous presented rules (*EDD*, *SPT*, *ATCS*, ...). For example we can have $\eta(j) = R_j(t,k)$ where $R_j(t,k)$ is defined above for the *ATCS* rule.

Algorithm 2: MMAS algorithm for scheduling problems

1 **Function** *MMAS*
  Input : A parallel machine scheduling problem $P|\beta|\sum T_j$ with a set of jobs $\mathcal{J}$, and a set of machines $\mathcal{M}$
  Parameters α, β, n_{ants}, τ_{min} and τ_{max}
  Output: A solution, i.e., a consistent assignment of all variables in $X = \{I_j, B_j, C_j | j \in \mathcal{J}\}$ (where I_j, B_j, and C_j represent the machine, start time, and completion time of job j, respectively.)
2 Initialize pheromone trails to τ_{max}
3 **while** *Stopping criterion not reached* **do**
4 $sol \leftarrow greedilyBuildSchedules(P, \alpha, \beta, n_{ants})$
5 Multiply every pheromone trail by $(1 - \rho)$
6 Reward pheromone trails according to *sol*
7 **if** *A pheromone trail is lower than* τ_{min} *(resp. larger than* τ_{max}*)* **then**
8 Set it to τ_{min} (resp. τ_{max})
9 **return** the best constructed solution
10 **Function** $greedilyBuildSchedules(P, \alpha, \beta, n_{ants})$
  Input : A scheduling problem P with jobs $\mathcal{J}$ and machines $\mathcal{M}$
  Parameters α, β, n_{ants}
  Output: A solution
  `/* Greedy randomised construction of one solution */`
11 $bestSol \leftarrow \perp$
12 **for** $k \in [1, n_{ants}]$ **do**
13 $Cand \leftarrow \mathcal{J}$
14 **while** $Cand \neq \emptyset$ **do**
15 choose a machine $i \in \mathcal{M}$ according to some heuristic
16 choose $j \in Cand$ w.r.t. probability $p(j) = \frac{[f_\tau(j)]^\alpha \cdot [\eta(j)]^\beta}{\sum_{j' \in Cand} [f_\tau(j')]^\alpha \cdot [\eta(j')]^\beta}$
17 assign i to I_j, and assign the smallest consistent values to B_j and C_j
18 $Cand \leftarrow Cand \setminus \{j\}$
19 **if** *The obtained schedule is better than bestSol* **then**
20 $bestSol \leftarrow$ the obtained schedule
21 **return** *bestSol*

The pheromone factor $f_\tau(j,i)$ represents the learned desirability of scheduling j on i and its definition depends on the used pheromone structure. For example when considering the *job* structure, we have $f_\tau(j,i) = \tau(j', j)$, where j' is the last job of i. When considering the *position* structure $f_\tau(j,i) = \tau(j, i, n)$ where n is the number of jobs on i plus one.

α and β are two parameters that are used to balance these two factors. In particular, when $\alpha = 0$, we obtain a *GRS* procedure, and when $\alpha = 0$ and β is very large, we obtain a greedy procedure.

Lines 5 to 8 are used to modify pheromone trails according to solutions computed during the last cycle (*i.e.*, according to the last n_{ants} constructed solutions) or according to solutions built since the beginning of the algorithm. Different methods are used to update pheromone trails.

In *MMAS*, three parameters are used : τ_{max}, τ_{min} and $\rho \in [0, 1]$. The pheromone trails are updated in two steps. First, every pheromone trail is decreased by multiplying it with $1 - \rho$. ρ is a parameter which controls the speed of intensification: the larger ρ, the quicker search is intensified towards the best solutions found recently. In a second step, pheromone trails associated with the best solution among the n_{ants} last computed solutions are increased in order to

Algorithm 3: Local search algorithm

Input : A parallel machine scheduling problem $P|\beta|\sum T_j$ with a set of jobs $\mathcal{J}$, and a set of machines $\mathcal{M}$, a neighborhood $\mathcal{N}$

Output: A solution, i.e., a consistent assignment of all variables in $X = \{I_j, B_j, C_j | j \in \mathcal{J}\}$ (where I_j, B_j, and C_j represent the machine, start time, and completion time of job j, respectively.)

1 $x_{incub} \leftarrow$ Compute initial solution
2 **while** *Stopping criteria are not met* **do**
3   Choose a neighbor $x_n \in \mathcal{N}(x_{incub})$
4     $x_{incub} \leftarrow x_n$
5 **return** the best solution found so far

increase the probability of selecting the components of this solution in the next constructions. Finally we ensure that each pheromone trail belongs to $[\tau_{min}, \tau_{max}]$.

There exist several variants of MMAS [GPG02, DS04]. In this thesis, we consider a variant where τ_{min} and τ_{max} are given parameters, and the best solution of the current cycle is rewarded. More precisely, let A_c (resp. A^*) be the best solution found among the n_{ants} last computed ones (resp. since the beginning of the search), every pheromone trail associated with A_c is increased by $(A_c - A^*)/A^*$.
Another strategy consists in setting τ_{max} to $1/A^*$, τ_{min} to $\tau_{max}/5$ and the increase is equal to $1/A_c$. In particular in that case, τ_{max} and τ_{min} evolve during the search (as A^* is updated each time a new best solution is found).

Finally it is worth mentioning that, in some variants, one can apply a local search (sec. 2.3) on the best solution found during the cycle (line 5) before updating the pheromone trails [DS04]. This local search may also be applied every k cycles with k a parameter.

2.3 Local Search

Local Search iteratively improves a solution by iteratively modifying some values. The local search algorithm is depicted in Algo. 3.

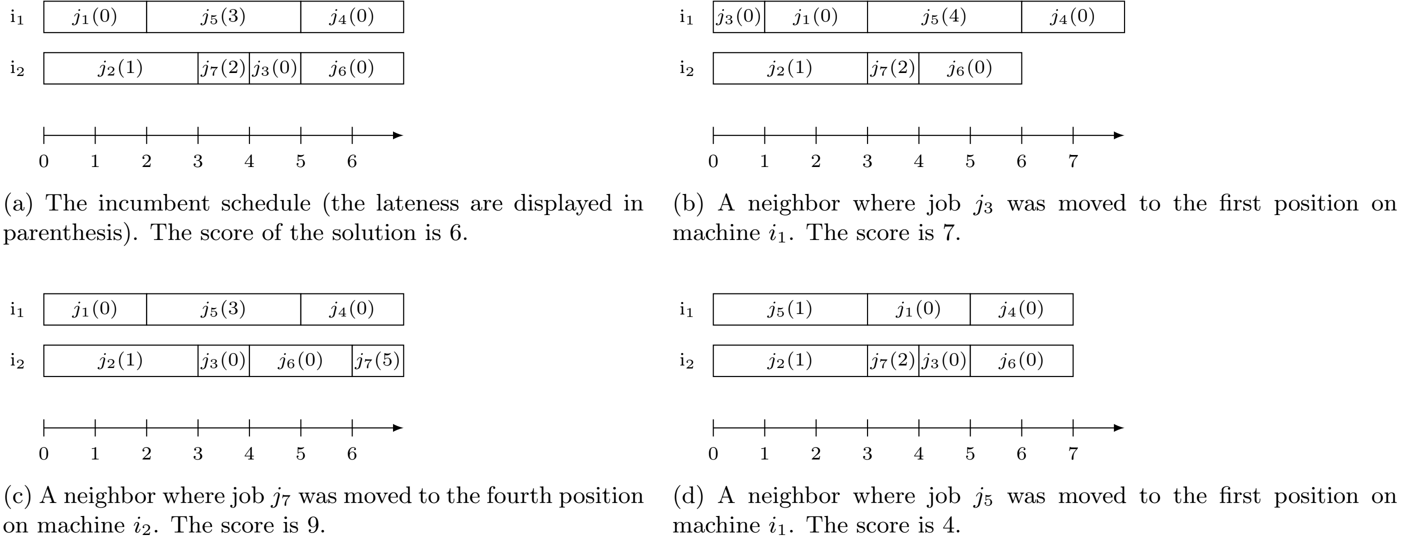
The algorithm consists of three steps. In a first step an initial solution is constructed. In a second step, we repeatedly choose a neighbor in the neighborhood of the current solution. We repeat this operation until the stopping criteria (maximum number of iteration reached, time limit reached, ...) are met. Finally, we return the best solution found so far. In the following subsection we will describe more precisely these steps.

2.3.1 Construction of an initial solution

Local search acts on an incumbent solution. However, in order to start the local search algorithm one needs to compute a first solution. To do so, we often use greedy approaches (*i.e.*, dispatching rules for scheduling problems, as described in 2.1). The main advantage of greedy approaches is that they often compute "good" solutions in a small amount of time. Hence the time saved during the construction of the initial solution can be spent later on the neighborhood walk.

2.3.2 Neighborhood

The idea of neighborhood is similar to the idea of gradient for continuous problems. Given a differentiable function f which we want to minimize, and a point x, the gradient of f evaluated in x indicates how f behaves around x. With discrete problems, it is not possible to compute gradients. Hence, in order to evaluate how the objective function behaves around our incumbent solution, we use *neighborhoods*. A neighborhood $\mathcal{N}$ is a function that associates a subset of the



(a) The incumbent schedule (the lateness are displayed in parenthesis). The score of the solution is 6.

(b) A neighbor where job j_3 was moved to the first position on machine i_1. The score is 7.

(c) A neighbor where job j_7 was moved to the fourth position on machine i_2. The score is 9.

(d) A neighbor where job j_5 was moved to the first position on machine i_1. The score is 4.

Figure 2.2: Three neighbors of a solution in the *insert* neighborhood

set of solutions to an incumbent solution. More formally, it is a function

$$\begin{array}{rccc} \mathcal{N}: & \mathcal{S} & \rightarrow & \mathscr{P}(\mathcal{S}) \\ & s & \mapsto & \mathcal{N}(s) \end{array}$$

where $\mathcal{S}$ is the set of all the solutions of the problem (where a the notion of *solution* is described in section 1.1).

Neighborhoods depend on the problem at hand and must be defined for each problem to solve. For scheduling problems, several neighborhoods exist. Besides, *neighborhoods* are often described in terms of operators. An operator is a function that transforms a solution into another solution.

The *insert* neighborhood (denoted $\mathcal{N}_R$) can be described as follows: given a schedule, all its neighbors are the schedules that are obtained by selecting a job and moving it on another machine or moving it elsewhere on its machine. Fig 2.2 shows some neighbors of a given schedule. This neighborhood's size is in $\mathcal{O}(n(n+m))$ where n is the number of jobs, and m is the number of machines. Indeed, for each job j in the incumbent schedule, we can get a neighbor by moving j before each other job (n possible choices) or at the machine's last position (m possible choices).

Another neighborhood, named *insert most late job* (denoted $\mathcal{N}_{RMLJ}$), consists in selecting the job which has the greatest tardiness in the current schedule and moving it elsewhere.
Given a solution $s \in \mathcal{A}$, the neighborhood defined by $\mathcal{N}_{RMLJ}$ is included in the one defined by $\mathcal{N}_R$ ($\mathcal{N}_{RMLJ}(s) \subset \mathcal{N}_R(s)$). The size of this neighborhood is equal to $\mathcal{O}(n+m)$.

Another neighborhood is the one named *swap* (denoted $\mathcal{N}_S$). Given a schedule, all its neighbors are the schedules that are obtained by selecting a job and swapping it with another job. The size of this neighborhood is in $\mathcal{O}(n^2)$.

There exist many neighborhoods for scheduling problems. [LBG91, Glo95] introduced the swap and the insert neighborhoods. Many other neighborhoods have been proposed since that time, some of them depending on the considered constraints and objective functions. For example, [KKJC02] proposes five neighborhoods for an unrelated parallel machine scheduling problem with setup-times and where objective functions consist of minimizing the sum of tardiness. For a similar problem, [LYL13] also proposes eight neighborhoods (for example, the *insert most late job* one). With the introduction of *Large Neighborhood Search* [Sha98], new larger neighborhoods have also been introduced. For example, [LG07] proposes three self-adapting neighborhoods (as an example, one of them consists of sorting all the jobs in an array according to their start times and then removing all the jobs whose index in the sorted array belongs to $\beta_W \cdot n, (\beta_w + \alpha_W) \cdot n$

and to reinsert them elsewhere, where n is the number of jobs of the problem, and β_W and α_W are two self-adapting parameters).

2.3.3 Choosing a neighbor in the neighborhood

In order to fully describe the algorithm depicted in Alg. 3, we must precise how we choose a neighbor in the neighborhood of an incumbent solution. Several strategy exists.

Local descent: The idea of local descent is to always select the best improving neighbor in the neighborhood. In some variants, if the neighborhood is large, the first improving neighbor met during the walk through the neighborhood is chosen. This algorithm quickly converges to a solution (as we always select the best neighbor, the objective function decreases quickly). However, its main drawback is that it get stuck into local minima. Indeed, when reaching a local optima, there is no improving solution in the neighborhood, and hence local descent stops.

Tabu search: Tabu search [GKL96] works in a similar fashion as local descent. It selects the best neighbor in the neighborhood. However, in order to avoid getting stuck in local optima, *tabu search* prevents the algorithm from going back to a solution it has already visited. This prohibition remains active until a given number of iterations are done (this number is called the *tabu list size*, and we call *tabu list* the list of all neighbors which are prohibited).

In practice, we do not forbid to visit the already visited solutions once again, but we forbid applying the reverse operator of the applied operator. As an example, if the current solution has been obtained by moving a job j from position l on machine i to position l' on machine i', we forbid to move job j back to position l on machine i. Forbidding operators (or more precisely reverse operators) instead of forbidding solutions has some advantages: it is smaller to memorize and it forbids more solutions which favors diversification (which leads to better results in practice).

Different strategies exist to fill the tabu list. For example, [LBG91] proposes seven strategies to fill tabu lists: preventing a job from going back to a position it has already occupied; preventing two jobs that have already been swapped to be swapped a second time (even if one of them has been moved between the two swaps); preventing a job from going to a position whose index is lower than an index the jobs has already occupied; ...

Since [GKL96], there has been much research on the method to improve it.

Tabu list's size: Under the questions at hand lies the one of the tabu list's size. The tabu list's size influences the algorithm. A big size will favor diversification because it will force the algorithm to get far from the previously found solutions. On the other hand, a short tabu list will favor intensification.
There exist variants of the algorithm [BT94] where the list's size is updated dynamically during the execution.

Aspiration criteria: Another idea to improve the algorithm concerns the *aspiration criteria*. As aforementioned, if we represent our neighborhood in terms of operators, the algorithm will sometimes ignore never evaluated solutions. It can be a vested investment to assess them and accept them only if they improve the current best-known solution. We called this method the aspiration criteria [Sal02].

Incremental search: a critical aspect of the tabu search lies in the walk through the neighborhood. Let's consider the neighborhood $\mathcal{N}_R$ for $P||\sum T_j$ problem. Let's assume that we are at iteration it of the tabu search algorithm. We evaluate the neighbor of our incumbent solution (x_{incub}) which is obtained by taking the job j which is on machine i_1 and moving it on machine i_2 (whatever the positions on i_1 and i_2). We observe that this neighbor doesn't improve our solution. We will consider all the neighbors of x_{incub} in order to find the best one: $x_{bestNeighbor}$. Now let's assume that $x_{bestNeighbor}$ is obtained from x_{incub} by moving job j' from position x on machine i_3 to position y on machine i_3. Now at iteration $it+1$, because nothing has changed on machines i_1 and i_2 between solutions x_{incub} and $x_{bestNeighbor}$, we know that moving job j from machine i_1 to machine i_2 will not improve our solution, and hence it is not necessary to evaluate

another time this neighbor.
Such incremental search can speed up the walk through the neighborhood.

Biased search: Another essential point when walking through the neighborhood lies in the walk's starting point. Let us consider the neighborhood $\mathcal{N}_R$. Suppose we always walk in the neighborhood in the same order: first, we try to move the job which is on position 1 on machine M_1; then, we consider the job which is on position 2 on machine M_1; ...; then the job at position 1 on machine M_2; then the job on position 2 on machine M_2; and so on. In that case, we will introduce a bias that will decrease our algorithm's performance. Instead, it is valuable to explore the neighborhood randomly.

2.4 Linear Programming

Linear Programming (LP) [V+15] aims at optimizing a linear objective function, subject to linear inequality constraints. The problem to solve must be modeled as:

$$\begin{array}{lll} & min & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \forall i \in [1,m] & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\ & \forall j \in [1,n] & x_j \geq 0 \end{array}$$

where x_j are variables and c_j, a_{ij} and b_i are constants.

Models are also often written using a matrix form:

$$\begin{array}{ll} min & \mathbf{c^T x} \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

Each inequality $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for a given i defines a hyperplane which separates the space into two half-spaces. The feasible region of the linear program (i.e. the set X such that $\mathbf{x} \in X$ satisfies $\mathbf{Ax} \leq \mathbf{b}$) is the intersection of many half-spaces. Fig 2.3 shows an example of the feasible region of a linear program.

As the objective function is linear and all x_j variables have continuous domains, we know that if the objective function has a minimum value on the feasible region, then it has this value on (at least) one of the extreme points [Mur83]. This property is at the root of a well-known algorithm used to solve linear programs: the simplex algorithm [Dan90].
The algorithm consists in two steps :

- Find an extreme point of the feasible region
- Move from extreme points to extreme points until the optimal value is found

The simplex algorithm is very efficient. However, it is essential to notice that the number of extreme points can be exponential with respect to the program's size. Hence this algorithm, even if it is very efficient in practice, is not polynomial.
Nevertheless, there exist polynomials algorithms [Kha80, Kar84], called interior-points methods, which can solve linear programs (with continuous variables). Thus linear programming with continuous variables is a polynomial problem.

Modeling: It is crucial to notice that for a given problem, there may exist several linear models that can solve the problem. According to the chosen model, the resolution can have different performances. In particular, it is possible to have a model for a problem which has an exponential number of variables or an exponential number of constraints. Thus, in that case, the model cannot be said to be polynomial. However, it does not mean that the problem is not polynomial.

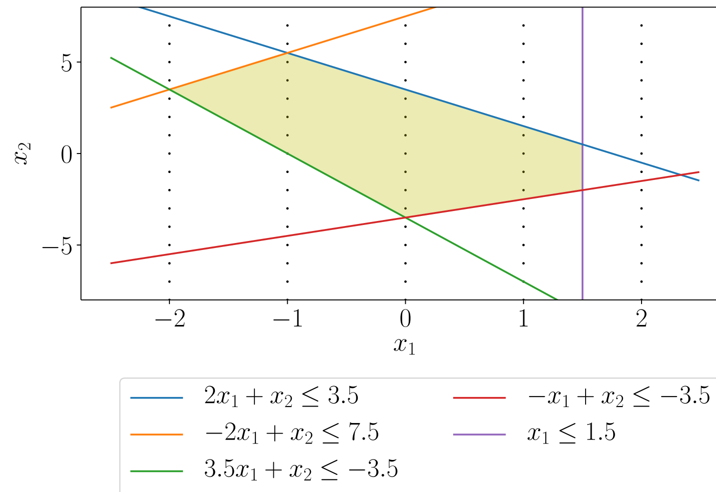


Figure 2.3: Feasible region of a linear program. There are 5 inequalities in the model, defining 5 half-spaces. The yellow region corresponds to the intersection of the 5 half-spaces.

Indeed, there may exist another model with continuous variables for the same problem, which uses a polynomial number of variables and constraints. Hence finding a model with polynomial numbers of variables and constraint means that the problem is polynomial, but the reverse is not true.

The model's choice significantly impacts the resolution's performance in a more general way. In the industrial world, linear solvers are often used as black-boxes (i.e., the choice of the algorithms used to solve (i.e., simplex methods, interior-point methods, ...) and their implementations is not made by the user). Hence, it is often the case that the person who wants to solve a problem only acts on the model itself and not on the solving process.

Column generation As aforementioned, many problems for which we know an exponential model exist. Let us assume that a model of such a problem has an exponential number of variables. In an optimal solution for this model, it is common to have many variables with a value equal to zero. The idea of column generation is to solve this model using two procedures that take turns.
In the first step, we consider only a subset of the variables, and we solve this partial problem. We call it the master problem. In a second step, we use a problem-dependent procedure (which we call the slave problem) to check whether there exists a variable not used in the master problem that can improve the solution if its value is not zero. If such a variable exists, we add it to the master problem and repeat the two processes.
Adding a new variable to the master model consists in adding a new column to the matrix $\mathbf{A}$. This is why this technique is called *column generation*.
In practice, we often add few columns before finding an optimal solution of the problem, and hence it is often the case that column generation solves problems that remain unsolved if we use linear programming without this technique. Here, it is essential to have a suitable procedure for the slave problem and a good model for the master problem.

Solvers As mentioned earlier, it is common not to implement the algorithms that solve linear programs and to use, instead, a solver with all these methods already implemented. Among the well-known solvers, we can cite CPLEX [Man87b], Gurobi [GO21], COIN-OR [Lou03].

2.4.1 Mixed Integer Linear Programming

Although linear programming with continuous variables is a polynomial problem, linear programming using integer variables (or mixing integer and continuous variables) is $\mathcal{NP}-hard$ in the general case. When all the variables of a model can take integer values, we speak about Integer Linear Programming (ILP). When only a subpart of the variables can take integer values (and the other can take real values), we speak about Mixed Integer Linear Programming (MIP).

Definition 2.4.1 (Mixed Integer Linear Programming (MIP))**.**
Mixed Integer Linear Programming (MIP) is a problem that aims at optimizing a linear objective function, subject to linear equality and linear inequality constraints, where some variables have continuous domains whereas others have integer domains.

$$
\begin{array}{lll}
 & min & \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \forall i \in [1,m] & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\
 & \forall j \in [1,n] & x_j \geq 0 \\
 & \forall j \in \mathcal{N} & x_j \in \mathbb{N}
\end{array}
$$

The subset $\mathcal{N} \subseteq [1,n]$ corresponds to the indexes of the variables which have integer domains.

Let us consider once again Fig 2.3. The black dots correspond to points where both x_1 and x_2 take integer values. We know that the objective function takes its minimum value on at least one extreme yellow region point. However, no extreme point of the region corresponds to an integer solution (i.e., a solution where both x_1 and x_2 take integer values). In such a case, the simplex algorithm will find assignments with continuous values for each variable, and hence such assignments will not be solutions of the problem (as some variables must have discrete values).

Branch and bound: A common technique to solve MIPs is *branch-and-bound* [BM07, LW66, Cla99]. Branch-and-bound consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating a branch's candidate solutions, the branch is checked against upper and lower estimated bounds on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.
The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search. Typically, a lower bound for a MIP may be obtained by solving the LP obtained by relaxing the integrality constraints.

2.4.2 Models for scheduling problems

Many models have been proposed for scheduling problems [Art12]. We will describe some of them in this section.

Assignment problem $P||C_{max}$

The model 2.4 describes a Linear Programming model for this problem. The model associates one binary variable X_{ij} for each machine $i \in M$ and for each job $j \in J$, whose value is equal to 1 if job j is assigned to machine i in the solution and 0 otherwise. One additional variable, named y, corresponds to the makespan's value. The constraint (1) states that each job must be assigned to at least one machine. The constraint (2) states that, for each machine, the makespan y must be greater than the sum of the durations of the jobs assigned to the machine. Constraints (3) and (4) specify the domains of the variables.

Release date and due date : $P|r_j|\sum T_j$

There exist two classical models for this problem.

$$
\begin{aligned}
&\text{Minimize } y \\
&\text{subject to } \sum_{i\in\mathcal{M}} X_{ij} \geq 1 && \forall j \in \mathcal{J} && (1)\\
&y \geq \sum_{j\in\mathcal{J}} p_j * X_{ij} && \forall i \in \mathcal{M} && (2)\\
&X_{ij} \in \{0,1\} && \forall j \in \mathcal{M}\ \forall i \in \mathcal{J} && (3)\\
&y \in \mathbb{N} && && (4)
\end{aligned}
$$

Figure 2.4: LP model for the assignment problem $P||C_{max}$.

$$
\begin{aligned}
&\text{Minimize } \sum_{j\in\mathcal{J}} T_j \\
&\text{subject to } \sum_{j\in\mathcal{J}} F_{0j} = \sum_{j\in\mathcal{J}} F_{j,n+1} = m && && (1)\\
&\sum_{k\in\mathcal{J}^0\setminus\{j\}} F_{kj} = \sum_{k\in\mathcal{J}^{n+1}\setminus\{j\}} F_{jk} = 1 && \forall j \in \mathcal{J} && (2)\\
&r_k F_{kj} \leq B_{kj} \leq h F_{kj} && \forall k,j \in \mathcal{J},\ k \neq j && (3)\\
&\sum_{k\in\mathcal{J}^{n+1}\setminus\{j\}} B_{jk} - \sum_{l\in\mathcal{J}^{n+1}\setminus\{j\}} (B_{lj} + p_l F_{lj}) \geq 0 && \forall j \in \mathcal{J} && (4)\\
&T_j \geq (\sum_{k\in\mathcal{J}^{n+1}\setminus\{j\}} B_{jk}) + p_j - d_j && \forall k \in \mathcal{J} && (5)\\
&F_{jk}, F_{0j}, F_{j,n+1} \in \{0,1\},\ B_{jk} \geq 0 && \forall j,k \in \mathcal{J}, j \neq k && (6)
\end{aligned}
$$

Figure 2.5: LP model for the scheduling problem $P|r_j|\sum T_j$. It is based on a flow model.

First model: The first one is depicted in Fig 2.5. It is adapted from the ones in [Art12, MSMA19]. It uses a flow. Such models are common to solve Vehicle Routing Problems.
We use subscript i to designate a machine (in the set $\mathcal{M}$), the subscript j to designate a job (in the set $\mathcal{J}$). As a reminder, p_j (resp. d_j, r_j) is used to designate the processing time of job j (resp. the due date of job j, the release date of job j). We consider two dummy jobs (denoted 0 and $n+1$) (whose durations are 0, whose release-dates are 0 and whose due-dates are h, the horizon). m units of flow leave the source job 0, and m units of flow enters the sink job $n+1$. We denote $\mathcal{J}^0 = \mathcal{J} \cup \{0\}$ and $\mathcal{J}^{n+1} = \mathcal{J} \cup \{n+1\}$.

We use three types of variables :

- F_{kj} for $(k,j) \subseteq \mathcal{J}^2$ whose value equals one if job j is positioned just after job k on a same machine. $F_{0j} = 1$ for $j \in \mathcal{J}$ indicates that j is the first job on its machine (otherwise $F_{0j} = 0$) and $F_{j,n+1} = 1$ for $j \in J$ indicates that j is the last job on its machine (otherwise $F_{j,n+1} = 0$)
- B_{jk} for $(j,k) \subseteq \mathcal{J} \times (\mathcal{J} \cup \{n+1\})$ which corresponds to the start time of job j if it precedes job k (or $n+1$)

- T_j for $j \in \mathcal{J}$ which corresponds to the tardiness of job j

We have the following constraints:

- Constraint (1) ensures that m units of flow leave the source and m units of flow enter the sink. This ensures that at most m machines are used.
- Constraint (2) ensures the flow conservation, meaning that jobs are processed one by one by any machine.
- Constraint (3) ensures that the start of each job is greater than its release date.
- Constraint (4) links flow and start times of jobs. In particular, it ensures that if job j follows job l, then the start time of j is greater than the start time of l plus the duration of l.
- Constraint (5) sets the value of the tardiness for each job j according to its start time.

Second model: The second one is depicted in Fig 2.6. It is adapted from the one used in [FL13].
We use the subscript l to designate a position on a machine (from 1 to n where n is the number of jobs). M is a large number.
We use three types of variables :

- X_{jil} for $j \in \mathcal{J}, i \in \mathcal{M}, l \in [1, n]$ whose value equals one if job j is assigned to position l on machine i and zero otherwise.
- $B_{i,l}$ for $i \in \mathcal{M}, l \in [1, n]$ which corresponds to the start time of the lth position on machine i
- T_j for $j \in \mathcal{J}$ which corresponds to the tardiness of job j

We have the following constraints:

- Constraint (1) ensures that each job is assigned to exactly one position on a machine.
- Constraint (2) ensures that at most one job is assigned to a given position.
- Constraint (3) ensures that the start time of the $(l+1)^{th}$ position on machine i is greater than the start time of the lth position plus the duration of the job assigned to that position.
- Constraint (4) ensures that the start time of the lth position on machine i is greater than the release date of j if job j is assigned to the lth position on machine i.
- Constraint (5) can be decomposed according to two different cases:
  - If X_{jil} is equal to one, the constraint can be written $T_j \geq B_{il} + p_j - d_j$, and we know that the job j is assigned to the lth position on machine i. Hence it ensures that the tardiness of job j is greater than the start time of job j plus its duration minus its due date.
  - If X_{jil} is equal to zero, as M is sufficiently large, the constraint can be written $T_j \geq -M'$ with M' a positive value, hence the constraint is automatically satisfied as $T_j \geq 0$
- Constraint (6) specifies the domains of the variables.

Speeds : $Q|r_j|\sum T_j$

We can easily adapt the model described in Fig 2.6 to the case where machines have different speeds. In such case, we can simply replace constraint (3) of Fig 2.6 by

$$B_{i,l+1} - B_{il} \geq \sum_{j \in \mathcal{J}} \frac{p_j}{s_i} X_{jil}$$

$$\text{Minimize} \quad \sum_{j \in \mathcal{J}} T_j$$

$$\begin{aligned}
\text{subject to} \quad & \sum_{i \in \mathcal{M}} \sum_{l=1}^{n} X_{jil} = 1 & \forall j \in \mathcal{J} & \quad (1) \\
& \sum_{j \in \mathcal{J}} X_{jil} \leq 1 & \forall i \in \mathcal{M}\ \forall l \in [1, n] & \quad (2) \\
& B_{i,l+1} - B_{il} \geq \sum_{j \in \mathcal{J}} p_j X_{jil} & \forall i \in \mathcal{M}\ \forall l \in [1, n-1] & \quad (3) \\
& B_{il} \geq \sum_{j \in \mathcal{J}} r_j X_{jil} & \forall i \in \mathcal{M}\ \forall l \in [1, n] & \quad (4) \\
& T_j \geq B_{il} + (X_{jil} - 1)M + p_j - d_j & \forall j \in \mathcal{J}\ \forall i \in \mathcal{M}\ \forall l \in [1, n] & \quad (5) \\
& X_{jil} \in \{0, 1\},\ B_{il}, T_j \geq 0 & \forall j \in \mathcal{J}\ \forall i \in \mathcal{M}\ \forall l \in [1, n] & \quad (6)
\end{aligned}$$

Figure 2.6: LP model for the scheduling problem $P|r_j|\sum T_j$.

Sequence dependent setup times : $Q|r_j, s_{jk}|\sum T_j$

The model described in Fig 2.6 can also be adapted to the case with setup-times. For such a case, we use additional boolean variables denoted Y_{jkil} for $\{j, k\} \subseteq \mathcal{J}\ i \in \mathcal{M}\ l \in [1, n-1]$. The value of Y_{jkil} is set to one if job j is assigned to position l on i and if job k is assigned to position $l+1$ on i, and zero otherwise. In other words Y_{jkil} is set to one if and only if both X_{jil} and $X_{ki,l+1}$ are set to one (*i.e.* $Y_{jkil} = X_{jil} \cdot X_{ki,l+1}$). Then, the difference with the model described in Fig 2.6 is on constraint (3) which is replaced by

$$B_{i,l+1} - B_{il} \geq \sum_{j \in \mathcal{J}} \left(\frac{p_j}{s_i} X_{jil} + \sum_{k \in \mathcal{J}} Y_{jkil} s_{jk} \right)$$

We must also add the following constraints:

$$Y_{jkil} \leq X_{jil}$$

$$Y_{jkil} \leq X_{ki,l+1}$$

$$Y_{jkil} \geq X_{jil} + X_{ki,l+1} - 1$$

in order to ensure that:

$$Y_{jkil} = X_{jil} \cdot X_{ki,l+1}$$

2.5 Constraint Programming

Constraint programming (CP) [RVBW06] allows a user to define a problem in a declarative way, by means of variables and constraints as described in section 1.1, and then to solve it by using generic search procedures.

2.5.1 Type of data and constraint

Using Linear programming, variables need to be numbers (real or integer), and both constraints and objective function are linear combinations of these variables. It is not the case for Constraint Programming. In CP, variables can take

values in an arbitrary set of symbols, and constraints (along with objective function) specify relations between variables that do not need to be linear combinations.

2.5.2 Constraint propagation

At the heart of Constraint Programming solving lies the notion of propagation. The propagation of a constraint c aims at filtering its variable domains by removing values that cannot belong to solutions. After this propagation step, the filtered domains are said to be *locally consistent*. Different propagation algorithms may be proposed for a same constraint, and these algorithms may achieve different levels of local consistency. Given two propagation algorithms P_1 and P_2 for a same constraint c, we say that P_1 is *stronger* than P_2 if, for every variable $x_i \in X(c)$, we have $D_1(x_i) \subseteq D_2(x_i)$ where D_1 and D_2 denote the filtered domains obtained by propagating c with P_1 and P_2, respectively, given the same initial domains.

In this section, we describe generalized arc consistency, which is the most famous local consistency.

Definition 2.5.1 (Generalized Arc Consistency (AC)).
Let be (X, D, C) a CSP, $c \in C$ a constraint, and $x_i \in X$ a variable.

- A value $v_i \in D(x_i)$ is consistent with c if there exists a valid tuple τ satisfying c such that $\tau[x_i] = v_i$. Such a tuple is called a *support* for (x_i, v_i) on c.
- The domain D is arc consistent on c for x_i if all values in $D(x_i)$ are consistent with c.
- The CSP (X, D, C) is arc consistent if D is arc consistent for all variables in X on all constraints in C.
- The CSP (X, D, C) is arc inconsistent if $\emptyset$ is the only domain tighter than D which is arc consistent for all variables on all constraints.

Historically, arc consistency is associated with binary CSPs and generalized arc consistency with non-binary CSPs while both definitions are perfectly the same.

Example 2.5.1. Let us consider the CSP introduced in Example 1.1.2. This CSP is not arc consistent because the value 1 for x_1 has no support on the constraint c_1 (because $\forall v \in D_{x_3}$ we cannot have $1 \geq v$).

When the domain of a variable x_i is not arc-consistent on a constraint c, we may ensure arc-consistency by removing inconsistent values from $D(x_i)$ and detect arc-inconsistency if the domain becomes empty. This domain filtering step is called *constraint propagation* and designing efficient propagation algorithms is a key point for solving CSPs.

2.5.3 Branch and propagate

Constraint propagation filters domains by removing values that cannot belong to solutions. However, it only ensures a local consistency, and constraint propagation must be combined with a systematic exploration of the remaining search space to actually find solutions (or prove inconsistency). We often use a branch and propagate principle:

- The initial problem is recursively decomposed into sub-problems by splitting variables' domains;
- For each sub-problem, constraints are propagated, and if one domain becomes empty then the sub-problem has no solution
- In a sub-problem, when all the variables' domains are reduced to singleton, then we have found a solution

There exist different strategies to choose how to split variables' domains and how to choose the order in which sub-problems are browsed. The interested reader should refer to [RVBW06] for further details.

2.5.4 Model and solvers

As for linear programming, a Constraint Programming's essential aspect lies in how one models a COP, which is then given to a solver. Among the most known CP Solvers, we can cite CP Optimizer [Man87a, LRSV18], ORTOOLS [PF19], Gecode [SLT06], Choco [JRL08].

2.5.5 Models for scheduling problems

In this section we will give some models for the scheduling problems described in section 1.2.5. Unless otherwise stated, the models are described using the *CP Optimizer* constraint language. More information can be found in [LR08, LRSV18, Bon17, Man87a]

Scheduling variables and constraints

In order to model scheduling problems, some specific variables and constraints are used. First there is the notion of interval variables. An interval variable represents an interval of time, i.e., a start time and an end time. Given an interval ι we denote B_ι its start time and C_ι its end time. An interval variable ι may be optional, and this is specified in CPO by adding $optional(\iota)$ to the model. When a variable is optional, it may either be present or absent. This is essential to model situations where jobs can be processed by different machines (or resources). In such a case, an interval variable is created for each machine, and, in a solution, only one variable is present and all the others are absent. The domain of the start time ranges from the earliest start time to the latest start time, and the domain of the end time ranges from the earliest end time to the latest end time. Decisions, made by propagation or during the search, will tend to reduce these ranges.

Among the constraints used in scheduling we can cite the *alternative* one. Given a set $\mathcal{S}$ of interval variables and an interval variable ι, the constraint $\iota = alternative(\mathcal{S})$ ensures that if ι is present then exactly one variable $\iota' \in \mathcal{S}$ is present (every other variable $\iota" \in \mathcal{S} \setminus \{\iota'\}$ is absent) and the start and end times of ι are equal to those of ι'.

Another used constraint is the $startMin$ constraint. It allows to specify that an interval cannot start before a given time. Hence, given an interval ι and a time $t \in [0, H]$, $startMin(\iota, t)$ ensures that ι starts after t (*i.e.*, $B_\iota \geq t$).

We must also mention the *intervalSequence* constraint. It allows to build a sequence of interval variables from a set of these variables. Given a set $\mathcal{S}$ of intervals, if we denote $\mathcal{S}_{pres}$ the set of intervals of $\mathcal{S}$ which are present (*i.e.*, $\mathcal{S}_{pres} = \{\iota' \in \mathcal{S} : \iota' \text{ is present}\}$), $intervalSequence(\mathcal{S})$ ensures that each interval $\iota \in \mathcal{S}_{pres}$ receives a different value from the set $\{1, \ldots, |\mathcal{S}_{pres}|\}$. This yields a total order on the execution of these intervals. We denote *seq* the sequence of intervals produced by $intervalSequence(\mathcal{S})$, and, for $\iota \in seq$ we denote $seq(\iota)$ the value received from the set $\{1, \ldots, |\mathcal{S}_{pres}|\}$. This constraint is often used in combination with the *noOverlap* constraint which is used to model disjunctive resources (such as machines in parallel machine scheduling). It forces the intervals variables of a sequence not to overlap. Given a set $\mathcal{S}$ of interval variables and *seq* the sequence such that $seq = intervalSequence(\mathcal{S})$, then $noOverlap(seq)$ ensures that for every pair of variables $\iota_1, \iota_2 \subseteq seq$, such that $seq(\iota_1) < seq(\iota_2)$, ι_1 ends before ι_2 starts (*i.e.*, $C_{\iota_1} \leq B_{\iota_2}$).
Along with a set of interval variables, one can specify a list of job types to the *intervalSequence* constraint. Doing so, each job has a type. Then, when combining with the *noOverlap* constraint, we can specify a setup-time between each type. Thus, given a set $\mathcal{S}$ of intervals, a set of types for each job $jobTypes$, and a value for the setup-time between each job type $setupTimes$, the combination of $seq = intervalSequence(\mathcal{S}, jobTypes)$ and $noOverlap(seq, jobTypes, setupTimes)$ ensures that for every pair of variables $\iota_1, \iota_2 \subseteq seq$, such that $seq(\iota_1) < seq(\iota_2)$, ι_1 ends $setupTimes_{type(\iota_1),type(\iota_2)}$ units of time before ι_2 starts (*i.e.*, $C_{\iota_1} + setupTimes_{type(\iota_1),type(\iota_2)} \leq B_{\iota_2}$) where $type(\iota)$ for an interval ι designate its type, and $setupTimes_{a,b}$ for a, b two types is the setup time which is due between these two types.

A very useful constraint to model hierarchical processes is the *Span* constraint, which states that a master interval should extend over all of time range covered by slave intervals. More precisely, given a set of intervals $\mathcal{S}$ (the slave intervals) and a (master) interval ι, the *span* constraint ensures that $B_\iota = \min_{\iota' \in \mathcal{S} : \iota' \text{ is present}} B_{\iota'}$ and $C_\iota = \max_{\iota' \in \mathcal{S} : \iota' \text{ is present}} C_{\iota'}$.

$$\text{Minimize} \quad \max_{j \in \mathcal{J}} \; endOf(a_j)$$

$$
\begin{aligned}
\text{subject to } & a_j^i = interval\,(p_j) & \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (1)\\
& optional(a_j^i) & \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (2)\\
& a_j = alternative(\{a_j^i : i \in \mathcal{M}\}) & \forall j \in \mathcal{J} \quad (3)\\
& S^i = intervalSequence(\{a_j^i : j \in \mathcal{J}\}) & \forall i \in \mathcal{M} \quad (4)\\
& noOverlap(S^i) & \forall i \in \mathcal{M} \quad (5)
\end{aligned}
$$

Figure 2.7: CP model for the assignment problem $P||C_{max}$.

Besides, the *intensity* constraint allows us to specify a percentage for a given period of time such that the time needed to complete a given interval during that period depends on that percentage. The *intensity* function represents an instantaneous ratio between the size and the length of an interval variable. For example, if the intensity is 100%, then the time needed to complete the interval is equal to its processing time; if it is 50%, then it will take twice more time to complete it; and so on. More precisely, given a function $f : [0, h] \rightarrow [0, 100]$, and an interval ι, $intensity(\iota, f)$ ensures that $C_\iota - B_\iota \geq \sum_{t=B_\iota}^{C_\iota} \frac{f(t)}{100}$.

We can also cite the *cumulative* constraints which are used to model the consumption of a resource by an interval variable. Cumulative constraints are used to model the fact that jobs require resources (*e.g.*, human skills or tools) and that these resources have limited capacities, *i.e.*, the sum of resources required by all jobs started but not ended must never exceed resource capacities. Given a set of intervals $\mathcal{S}$ such that each interval $\iota \in \mathcal{S}$ consumes con_ι units of the considered resource, and l the capacity of the resource, the cumulative constraint ensures that each time $t \in [0, h]$ the intervals consumption is lower than l, *i.e.*, $\sum_{\iota \in \mathcal{S} : \iota \text{ is present} \wedge b_\iota \leq t < c_\iota} con_\iota \leq l$

Assignment problem $P||C_{max}$:

The model in Fig. 2.4 can also be solved by *Constraint Programming* solvers (boolean variables are classical in CP Models and linear relations can also easily be modeled). However another way to see this assignment problem is through a *scheduling* point of view. In that case we use *interval* variables and we schedule them over machines (this model is closer to the model used in the following sections as it uses *interval* variables). Such a model is described in model 2.7. An interval a_j is associated with every job $j \in \mathcal{J}$, *i.e.*, a_j corresponds to the interval $[B_j, C_j]$, with B_j the start time of job j and C_j its end (or completion) time. An optional interval variable a_j^i is associated with every job $j \in \mathcal{J}$ and every machine $i \in \mathcal{M}$: if job j is executed on machine i, then $a_j^i = a_j$; otherwise a_j^i is absent). Finally, an interval sequence variable S^i is associated with every machine i to represent the total ordering of the present interval variables in $\{a_j^i : j \in \mathcal{J}\}$.

- (1) and (2) define the interval variable a_j^i whose length is equal to the processing time of job j ;
- Constraint (3) ensures that every job j is scheduled on exactly one machine;
- Constraint (4) defines the sequence of jobs on machine i;
- Constraint (5) ensures that at most one job is executed at a time on machine i

Release date and due date $P|r_j|\sum T_j$:

The model for this problem is similar to the one of the previous problem (Fig. 2.7). We only need to add a new constraint which specifies that interval a_j^i cannot start before the release date r_j of job j:

$$startMin(a_j^i, r_j) \quad \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (6)$$

The objective function is also different: now, instead of minimizing the makespan, we minimize the sum of all the jobs' tardiness:

$$\sum_{j \in \mathcal{J}} \max(0, endOf(a_j) - d_j)$$

Speed problem $Q|r_j|\sum T_j$:

In order to model speeds on machines, we only replace, in the previous model, the constraint (1):

$$a_j^i = interval\,(p_j) \quad \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (1)$$

by

$$a_j^i = interval\left(\frac{p_j}{s_i}\right) \quad \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (1)$$

Sequence dependent setup times problem $Q|r_j, s_{jk}|\sum T_j$:

In order to take into account the sequence dependent setup-times we need to modify constraints (4) and (5). Hence, we first need to replace

$$S^i = intervalSequence(\{a_j^i : j \in \mathcal{J}\}) \quad \forall i \in \mathcal{M} \quad (4)$$

by

$$S^i = intervalSequence(\{a_j^i : j \in \mathcal{J}\},\ jobTypes) \quad \forall i \in \mathcal{M} \quad (4)$$

this allows us to specify the type of each job (the sequence-dependent setup-times depend on the types of each job). The other modification consists in replacing

$$noOverlap(S^i) \quad \forall i \in \mathcal{M} \quad (5)$$

by

$$noOverlap(S^i,\ jobTypes,\ setupTimes) \quad \forall i \in \mathcal{M} \quad (5)$$

which allows us to specify the setup time which is due between each type.

This model is written using the constraints available in *CP Optimizer*. However, such constraints are not available in all *CP solvers*. For example, it is impossible to do so with the *ORTOOLS solver*.
For ORTOOLS, to model sequence-dependent setup times, it is necessary to represent all the assigned jobs in a complete directed graph (each vertex represents a job, and there is an arc between every two vertices, whose weight is equal to the duration of the setup-time due between the two jobs). Then, the only thing to do is to add a *circuit* constraint on that graph. Thus if an arc is included in the circuit, then the setup time between the two extremities of the arc must be applied.

Breaks problem $Q|r_j, brkdwn|\sum T_j$:

To model scheduled breakdowns, we use a function $openPeriod_i(t)$ whose value is equal to 100 if machine i is open at time t and zero otherwise (notice that the break periods are not the same on all the machines). Then we must add the following new constraint to the model used for the speed problem $Q|r_j|\sum T_j$

$$intensity(a_j^i, openPeriod_i) \quad \forall j \in \mathcal{J}, \forall i \in \mathcal{M} \quad (7)$$

$$\text{Min} \quad \sum_{j \in \mathcal{J}} \max(0, endOf(a_j) - d_j)$$

$$
\begin{aligned}
\text{s.t. } & a_j^i = interval\left(\frac{p_j}{s_i}\right) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (1)\\
& intensity(a_j^i, openPeriod_i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (2)\\
& optional(a_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (3)\\
& a_j = alternative(\{a_j^i : i \in \mathcal{M}\}) && \forall j \in \mathcal{J} && (4)\\
& setup_j^i = interval() && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (5)\\
& intensity(setup_j^i, openPeriod_i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (6)\\
& startMin(setup_j^i, r_j) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (7)\\
& optional(setup_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (8)\\
& presenceOf(setup_j^i) = presenceOf(a_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (9)\\
& startAtEnd(a_j^i, setup_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (10)\\
& cover_j^i = interval() && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (11)\\
& optional(cover_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (12)\\
& presenceOf(cover_j^i) = presenceOf(a_j^i) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (13)\\
& span(cover_j^i, \{a_j^i, setup_j^i\}) && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (14)\\
& S^i = intervalSequence(\{cover_j^i : j \in \mathcal{J}\}, jobTypes) && \forall i \in \mathcal{M} && (15)\\
& noOverlap(S^i) && \forall i \in \mathcal{M} && (16)\\
& lengthOf(setup_j^i) = \\
& \quad setupTimes[typeOfPrevious(S^i, cover_j^i, type[j])][type[j]] && \forall j \in \mathcal{J}, \forall i \in \mathcal{M} && (17)
\end{aligned}
$$

Figure 2.8: A CP model for the $Q|r_j, brkdwn, s_{jk}|\sum T_j$ problem

If the intensity is 0% on a given interval, then no amount of work can be done on that job during this period. Hence in our case, we set the intensity to 0% for a particular machine for all the jobs during the machine's breakdowns and to 100% when the machine is not on a break.

Breaks and sequence-dependent setup times problem $Q|r_j, brkdwn, s_{jk}|\sum T_j$:

As we have already models for the case *sequence-dependent setup times* and the case *breaks* a natural idea consists in combining these models (i.e. adding constraint (7) to the model used for $Q|r_j, s_{jk}|\sum T_j$). However, doing so does not lead to the expected result. Indeed, as explained in section 1.2.5, the setup-times cannot be executed during the breaks (as they correspond to human activities). For this reason, another model is needed. In our case, setup-times must be seen as sequence-dependent setup activities [Man87a, LRSV18]. The model is described in Fig 2.8.

For each job $j \in \mathcal{J}$ it uses three variables. The first one a_j^i is used to represent the interval corresponding to the execution of the job (if it is executed on machine i). The second one $setup_j^i$ is used to represent the setup time of job j (if executed on machine i). And finally the last one $cover_j^i$ is used to represent the whole job (setup time plus processing time). Hence the model can be explained as follows:

- (1) defines the interval variable a^i_j whose length is equal to the processing time of job j divided by the speed of the machine i ;
- (2) ensures that the processing of the jobs is stopped during the breaks on machines;
- (3) and (4) ensure that every job j is scheduled on exactly one machine;
- (5) defines the setup-time interval for job j (whose length is fixed by constraint (17)) ;
- (6) ensures that the setup-times are not executed during the breaks;
- (7) ensures that the setup-times do not start before the release date of jobs;
- (8) and (9) ensure that the setup-time for job j on machine i is only applied if job j is executed on machine i;
- (10) ensures that the setup-time interval of job j is executed just before the processing interval of the job;
- (11) defines the interval which covers the whole job j (representing its setup-time plus its processing time);
- (12) and (13) ensure that the whole interval for job j on machine i is only applied if job j is executed on machine i;
- (14) ensures that $cover^i_j$ starts when $setup^i_j$ starts and ends when a^i_j ends;
- (15) defines the sequence of jobs on machine i (specifying the type of each job);
- (16) ensures that at most one job is executed at a time on machine i;
- (17) specifies the length of the setup-time interval of each job. $typeOfPrevious(S^i, cover^i_j, type[j])$ corresponds to the type of the job which is just before job j on machine i (or $type[j]$ if j is the first job of the machine). $setupTimes[t_1][t_2]$ is the length of the setup type if we execute a job of type t_2 just after a job of type t_1. (if job j is the first job of the machine, we say that the previous type is the same as the one of j, and hence the length of the setup time interval is zero.);

It is noticeable that this model is much more complicated than the previous ones.

2.6 Discussion

In this chapter we introduced several state-of-the-art methods used to solve scheduling problems. The ways these methods solve problems are totally different. *ACO* is an incomplete constructive approach, which repeatedly constructs new solutions starting from empty solutions. Furthermore it learns from its previous constructions in order to improve the next ones. *Tabu search* is an incomplete perturbative approach which always works on the same solution and makes many little modifications to this solution in order to improve it. ACO and tabu search quickly compute solutions, but there is no guarantee on the quality of these solutions. ILP and CP are exact approaches which are able to find optimal solutions and prove optimality, at the expense of exponential time complexities in the worst case. When using *Linear Programming* and *Constraint Programming* the user must define a problem in a declarative way, by means of variables and constraints. The model is defined using a restricted set of variables and constraints. For example, when using Linear Programming, all the constraints must be linear. In both cases, models are then solved using generic search procedures. We illustrated each of these methods on the scheduling problems considered in this thesis. In particular, for *ACO* we gave several heuristic approaches (EDD, LPT, MS, ATCS, ...) and two well-known pheromone structures (*Jobs* and *Position* trails). For *tabu search* we described some neighborhoods and some strategies to fill tabu lists. Concerning *Linear* and *Constraint programming*, we listed several models used to solve the scheduling problems presented in the previous chapter.

It is crucial to mention that none of these methods outperforms the others on all COPs. Hence, each methods has its own strengths and weaknesses, and depending on the problem, some are more adapted than others.
We have restricted our attention to the most well known approaches for solving scheduling problems. There exist

many other methods for solving COPs: *Genetic algorithms* [SD08, Mit98], *Variable Neighborhood Search* [HM14], *Large Neighborhood Search* [PR10], SAT Solvers [GPFW96, HJS+18], ...

In the next chapters we will introduce a new scheduling problem, and we will show how the methods presented here can be adapted to this new scheduling problem. We will also present a new method which can be applied on this new problem as well as on the problems presented so far.

Part II

The Group Cumulative Scheduling Problem

Chapter 3

Study of the GCSP

Contents

3.1 Definition **61**

3.2 Related problems **63**

3.2.1 Cumulative scheduling problems 63

3.2.2 Open Stack Problems 63

3.2.3 Hierarchical scheduling problems 64

3.3 Complexity **66**

3.3.1 Classical cumulative constraints 66

3.3.2 Group cumulative constraint 67

3.4 Discussion **71**

In this chapter we introduce a new scheduling problem, called the *Group Cumulative Scheduling Problem*. In section 3.1 we formally introduce the problem. We describe how it can be modeled and give an example of an instance of this problem to illustrate. In section 3.2 we describe related problems, especially *Cumulative Scheduling Problems*, *Open Stacks Problems* and *Hierarchical scheduling problems*. We highlight the common features and the differences between these problems. In section 3.3 we study the complexity of this problem from a theoretical point of view. We show that converting a *list-schedule* into a schedule is an $\mathcal{NP}-complete$ problem when considering the *Group Cumulative* constraint whereas it is not when we do not consider this constraint.

3.1 Definition

In the *Group Cumulative Scheduling Problem (GCSP)*, jobs are partitioned into groups. The start (resp. end) time of a group is defined as the smallest start time (resp. largest end time) among all its jobs. A group is said to be *active* at a time t if it is started and not ended at time t. The number of active groups must never exceed a given limit.

Definition 3.1.1 (*Group Cumulative Scheduling Problems*)**.**
The *Group Cumulative Scheduling Problems* is defined as follows:

- **Input Data:** A tuple $(\mathcal{M}, \mathcal{J}, \mathcal{P}, l)$ such that:
  - $\mathcal{M}$ is a set of machines;
  - $\mathcal{J}$ is a set of jobs, such that each job $j \in \mathcal{J}$ has a processing time $p_j \in \mathbb{N}$;
  - $\mathcal{P}$ is a partition of $\mathcal{J}$ in $|\mathcal{P}|$ groups (such that each job $j \in \mathcal{J}$ belongs to exactly one group $g \in \mathcal{P}$). Each group $g \in \mathcal{P}$ has a due date d_g ;

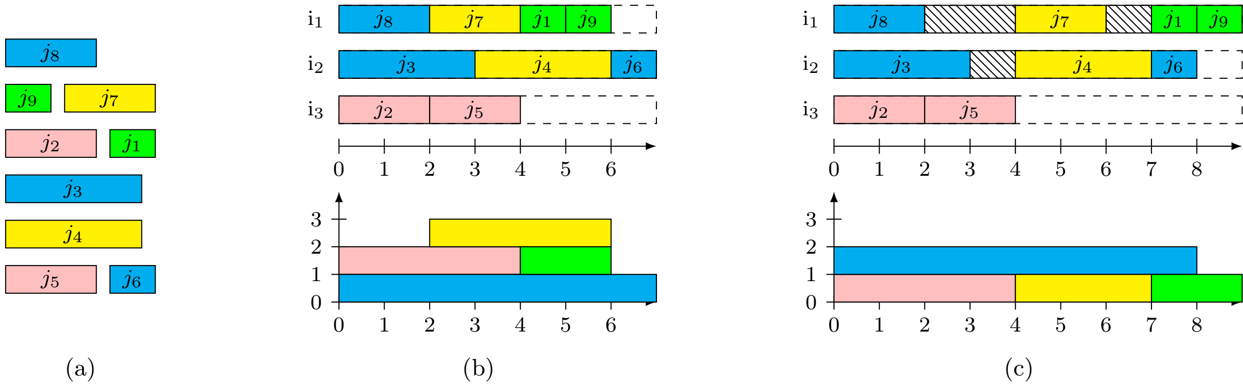


Figure 3.1: Schedule examples.
(a) A set $\mathcal{J}$ of 9 jobs and a partition $\mathcal{P}$ of $\mathcal{J}$ in 4 groups represented by colors
(b) Example of schedule on 3 machines which violates GC when $l = 2$ (there are 3 active groups from time 2 to time 6, as displayed on the bottom of the figure)
(c) Example of schedule on 3 machines which satisfies GC when $l = 2$ (an idle time is added between j_8 and j_7 and between j_3 and j_4 to wait the end of the pink group before starting the yellow group)

– l is the maximum allowed number of active groups at any time t

– $h \in \mathbb{N}$ the time horizon (and $\mathcal{H} = \{0, ..., h\}$)

- **Variables:** For each job $j \in \mathcal{J}$, variables I_j, B_j and C_j represent the machine that processes j, the start time of j and the completion time of j, respectively. For each group $g \in \mathcal{P}$ B_g and C_g the start time of g and the completion time of g, respectively The domains of these variables are $\mathcal{D}(I_j) = \mathcal{M}$ and $\mathcal{D}(B_j) = \mathcal{D}(E_j) = \mathcal{D}(B_g) = \mathcal{D}(E_g) = \mathcal{H}$.
- **Constraints:**

$$\forall j \in \mathcal{J},\ C_j = B_j + p_j \tag{3.1}$$

$$\forall g \in \mathcal{P},\ B_g = \min_{j \in g} B_j \tag{3.2}$$

$$\forall g \in \mathcal{P},\ C_g = \max_{j \in g} C_j \tag{3.3}$$

$$\forall \{j_1, j_2\} \subseteq \mathcal{J},\ I_{j_1} = I_{j_2} \Rightarrow C_{j_1} \leq B_{j_2} \vee C_{j_2} \leq B_{j_1} \tag{3.4}$$

$$\forall t \in \mathcal{H},\ |\{g \in \mathcal{P} | B_g \leq t \leq C_g\}| \leq l \tag{3.5}$$

  Constraint (3.1) relates the end time of a job to its start time and processing time. Constraints (3.2) and (3.3) relate the start and end time of a group to the start and end times of its jobs. Constraint (3.4) ensures that jobs assigned on a same machine do not overlap. Constraint (3.5) ensures that the number of active groups never exceeds the limit l. This constraint is called the Group Cumulative (GC) constraint.
- **Objective function:** Minimizing $\sum_{g \in \mathcal{P}} T_g$ with $T_g = \max(0, C_g - d_g)$

In Fig. 3.1, we display two examples of schedules: one is not solution of the GCSP, and one that is solution. We can consider different variants of the *GCSP* if we consider or not sequence-dependent setup-times, scheduled breakdowns on machines, precedence between jobs... In order to indicate that a scheduling problem is a variant of the *GCSP*, we use the notation *GC* in the field β of the Graham notation $(\alpha|\beta|\gamma)$.
Furthermore, for the *GCSP*, the objective is to minimize the sum of tardiness over the groups (instead of the sum of jobs' tardiness in other scheduling problems presented so far). We denote this new objective function $\sum_{g \in \mathcal{P}} T_g$ in the field γ of

the Graham notation.
Hence, $P|GC|\sum T_g$ corresponds to the GCSP; $P|GC, r_j, s_{jk}|\sum T_g$ corresponds to the GCSP with sequence-dependent setup-times and release-dates,...

Our industrial application for the *GCSP* is described precisely in chapter 5. In a word, our application deals with order preparation. The considered jobs are tasks which must executed to complete orders. Jobs of a same order are put on a same pallet to be shipped together. Hence, as we start the jobs of an order, we put a pallet on the shop floor, and the different jobs fill the pallet. The pallet remains on the floor until all its jobs are done. And, the physical space available for pallets is limited.

3.2 Related problems

We describe in this section the links with three other problems: *Cumulative scheduling problems*, *Open Stacks Problems* and *Hierarchical scheduling problems*.

3.2.1 Cumulative scheduling problems

Cumulative constraints [AB93, BB18, OQ13, NS03, Bon17] are used to model the fact that jobs require resources (*e.g.*, human skills or tools) and that these resources have limited capacities, *i.e.*, the sum of resources required by all jobs started but not ended must never exceed resource capacities. More formally, given a set $\mathcal{J}$ of jobs, and a cumulative resource of capacity R, such that each job $j \in \mathcal{J}$ consumes λ_j units of the cumulative resource, we say that the resource is satisfied if and only if

$$\forall t \in [0, h] \sum_{\substack{j \in \mathcal{J} \\ B_j \leq t < C_j}} \lambda_j \leq R$$

The *GCSP* can be seen as a generalization of cumulative scheduling problems where the jobs' consumption is equal to one ($\forall j \in \mathcal{J}, \lambda_j = 1$). Indeed, a cumulative scheduling problem with a resource of capacity R, is a special case of the *GCSP* where we have one group g for each job $j \in \mathcal{J}$ which contains only j, and where we set the limit of the group cumulative resource to R. Furthermore, we could consider a variant of the *GCSP* where each group $g \in \mathcal{P}$ consumes λ_g units of the group cumulative constraint. In such a case, the *GCSP* would be a generalization of cumulative scheduling problems (even in the cases where $\lambda_j \neq 1$). The study of such a variant is let as future work, and is out of the scope of this thesis.

3.2.2 Open Stack Problems

Definition and example

Open Stack Problems are not considered as scheduling problems. They deal with sequencing operations (named patterns in the terminology), but there is no need to define start (or end) times for these operations (start and end times can easily be derived from the sequence of operations). The *Minimizing Open Stack Problem* (MOSP) can be formulated as follows [AS09, LY02, Yan97]:
We are given a set of piece types I, and a set of patterns J, where a pattern is a subset of piece types ($\forall j \in J,\ j \subseteq I$). We can define a *piece-pattern relationship* by an $I \times J$ binary matrix $P = \{p_{ij}\}$ where $p_{ij} = 1$ if pattern j contains piece type i and 0 otherwise. The objective if to find a sequence of the patterns (i.e. a permutation π of the patterns) such that the maximum number of open stacks is minimized. To complete the definition we must specify what an open stack is. Given a permutation π of the patterns, we can define an *open stacks versus cutting instants matrix* $Q^\pi = \{q^\pi_{ij}\}$ where

$$q^\pi_{ij} = \begin{cases} 1, & \text{if } \exists x, y \in J \text{ s.t. } \pi(x) \leq j \leq \pi(y) \text{ and } p_{ix} = p_{iy} = 1 \\ 0, & \text{otherwise} \end{cases}$$

Hence, the Q^π matrix has one line for each piece type and one column for each pattern. The columns are in the same order as in the permutation π. Given a line (i.e. a piece type i) there are ones on that line between the first column x which represents a pattern such that $p_{ix} = 1$ and the last column y which represents a pattern such that $p_{iy} = 1$, and there are zeros on the line everywhere else. Hence the ones are consecutive for columns in that matrix. We say that a stack is open for a piece type i at an instant $t \in [1, |J|]$ for a permutation π if $q^\pi_{it} = 1$. Hence the maximum number of simultaneous open stacks in a permutation π is

$$Z^\pi = \underset{j\in[1,|J|]}{max} \sum_{i\in[1,|I|]} q^\pi_{ij}$$

And the objective of the problem is

$$\min_\pi Z^\pi$$

Fig 3.2 shows an example of such problem.

MOSP can represent the context of woodcutting with limited storage space around the saw machine. In such a context, there are different panel types. A pattern consists of pieces cut into several panel types. A stack is open for every new panel type, and it remains open until the last piece of that panel type is cut. However, the space around the saw is limited, and hence we want to minimize the number of simultaneously open stacks.

A problem close to the MOSP is the *Minimization of Order Spread Problem* (MORP, also called *pattern allocation problem* or *cutting sequencing problem*). The definition of the MORP is similar to the one of the MOSP. However, instead of minimizing the maximum number of open stacks at any time, the objective is to minimize the mean *lifetime* of open stacks (where the lifetime of a stack is defined as the difference between the time at which the stack is closed and the time at which it was opened). It is similar to minimizing the sum over all the stacks of their lifetime.
If we consider once again Fig 3.2, the solution in part 3.2c the total lifetime of all the stacks is $5+3+3+3+6+6+3 = 29$ whereas the total lifetime of the solution in 3.2d is $2 + 3 + 2 + 2 + 2 + 4 + 2 = 17$.

It is worth mentioning that both MOSP [LY02] and MORP [GGJK78] are $\mathcal{NP}-Complete$.

Link with the GCSP

Although both GCSP and Open Stacks Problems (OSP) model real industrial problems where the physical space for underway tasks is limited, they have significant differences in terms of resolution. For the similarities, in both problems, there are entities (patterns in the OSP and jobs in the GCSP) which must be 'scheduled' and which belong to sets (the sets are piece type for the OSP and the groups for the GCSP). Moreover, these sets consume one unit of a capacity-limited resource during all the time between the start time of their first entity until the end time of their last entity.
However, there are two main differences. First of all, in OSP, one must find a unique permutation, whereas, in GCSP, one must divide the jobs over different machines, find permutations for each machine, and find each job's start time to satisfy the GC. Furthermore, in the GCSP, each job belongs to exactly one group that consumes one unit of the resource, whereas in the OSP, a pattern belongs to several piece types, and each piece type consumes one unit of the resource.
For the terminology, for the *GCSP* we say that a group is *active* at a time t if t is between the start time and end times of the group. For the OSP, we say that piece types are *open* between their start and end times.

3.2.3 Hierarchical scheduling problems

Hierarchical scheduling problems (HSP) [PPR18, PPR19b, PPR19a] are defined by tuples $(\mathcal{R}, \mathcal{A}, \mathcal{C})$ such that:

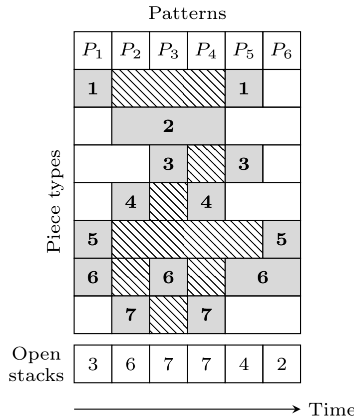
- $\mathcal{R}$ is a set of disjunctive resources;
- $\mathcal{A}$ is a set of activities which must be realized (also called atomic task). Each task is characterized by its duration du_a and by the resources it consumes during its execution $\mathcal{R}_a \subset \mathcal{R}$
- $\mathcal{C}$ is a set of composite tasks, where each composite task c is characterized by

$P_1 = \{1, 5, 6\}$
$P_2 = \{2, 4, 7\}$
$P_3 = \{2, 3, 6\}$
$P_4 = \{2, 4, 7\}$
$P_5 = \{1, 3, 6\}$
$P_6 = \{5, 6\}$

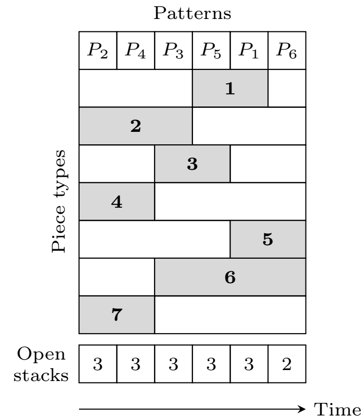
(a) Content of each pattern. For example pattern P_1 contains item types 1, 5 and 6

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(b) Representation of the *piece-pattern relationship*



(c) A solution where the sequence of patterns is $(1, 2, 3, 4, 5, 6)$. During P_3 seven stacks are open simultaneously, which is the maximum reached. The stack of piece type 4 is open while executing patterns 2, 3 and 4 for example, even if piece type 4 is not in pattern 3. This is a representation of the matrix Q^π: white cells correspond to 0 in the matrix, whereas gray or hatched cells correspond to 1. A gray cell corresponds to a value where $p_{ij} = 1$ whereas hatched cells for a line i correspond to a value of a pattern z such that $p_{iz} = 0$ and such that there exist two patterns x and y with $\pi(x) < \pi(z) < \pi(y)$ and $p_{ix} = p_{iy} = 1$



(d) A solution where the sequence of patterns is $(2, 4, 3, 5, 1, 6)$. The maximum number of open stacks is equal to 3. The stack corresponding to piece type 6 is open while executing patterns 3, 5, 1 and 6. As pattern P_3 contains three piece types, a lower bound on the objective function is 3. Hence this solution is optimal.

Figure 3.2: A representation of an instance of the *minimizing open stack problem*

– A subset of tasks $SubTsk_c \subset \mathcal{C} \cup \mathcal{A}$

– A set of resources it consumes during its execution $\mathcal{R}_c \subset \mathcal{R}$ (where its execution starts when its first subtask starts and ends when its last subtask ends)

– A set of acyclic precedence constraints $\mathcal{P}_c$ between the subtasks of c

In order to limit the study to interesting problems, the following assumptions are often made:

- The graph of tasks decomposition is acyclic. More formally, this graph is an oriented graph where we have one node for each task in $\mathcal{A} \cup \mathcal{C}$ and an arc between each composite task $c \in \mathcal{C}$ and each of its subtask $\tau \in SubTsk_c$.
- Each atomic or composite task is a subtask of at most one task. This implies that the graph of tasks decomposition is a forest.
- A resource consumed by a composite task $c \in \mathcal{C}$ is not consumed by any other of its descending task τ. More formally, for each composite task $c \in \mathcal{C}$ and for each task τ such that there exists a path between c and τ in the graph of tasks decomposition, $\mathcal{R}_c \cap \mathcal{R}_\tau = \emptyset$.

A hierarchical scheduling problem respecting these three additional constraints is said to be *well-formed*.

The GCSP can be modeled using this formalism, where atomic tasks are the jobs, and each group is a composite task (its subtasks being the group's jobs). Hence the two problems are quite close.
However, as stated above, HSP's resources are disjunctive, and in the GCSP, the resource is a cumulative one. Furthermore, the tasks in HSP consume resources which are given as input (set $\mathcal{R}_a$), whereas, in GCSP, each job must consume one unit of a disjunctive resource (the machines), but the resource it uses is a decision variable (not an input). Moreover, there are in HSP precedence constraints that are not present in GCSP. Besides, these precedence constraints and the fact that resources are disjunctive are exploited in algorithms presented in the articles mentioned above. For example, they are exploited to approximate a composite task's duration by saying that it approximately lasts the sum of the duration of its subtasks, which are linked by precedence constraint. Alternatively, it approximately lasts the sum of the tasks' duration, which uses the same disjunctive constraint. However, in the GCSP, a group can contain m jobs in a problem with m machines. In such a problem, the duration of the group can be the duration of its longest job (if we schedule them in parallel with the same start time) or the sum of the duration of its jobs (if they are all on the same machine), or something even greater (if there are other jobs or idle time between the jobs). Hence, approximating a group's duration with precision is difficult in the GCSP, and the HSP techniques may not be efficient when applying them to the GCSP.

3.3 Complexity

As aforementioned, the scheduling problem $P|r_j|\sum T_j$ and all its generalizations are NP-hard. However, if we know the ordered list of jobs that must be scheduled on every machine, then we can compute the start times that minimize the tardiness sum in polynomial time [Sch96, HK00]. More precisely, a *list schedule* is a set of m ordered lists $l_1, \dots l_m$ such that each job of $\mathcal{J}$ occurs in exactly one list. Given a list schedule, we greedily compute optimal start times: for each machine i, we consider jobs according to the order defined by l_i and schedule each of these jobs as soon as possible. Therefore, solving scheduling problems without resource constraints (such as the ones presented in section 1.2.5) amounts to finding the best list schedule (and start times are derived in polynomial time from these lists).

Let us now consider the cases where we add a classical cumulative constraint (Section 3.3.1), or a GC constraint (Section 3.3.2).

3.3.1 Classical cumulative constraints

If we add a classical cumulative constraint to the scheduling problem $P|r_j|\sum T_j$, the problem of computing the start times which minimize the sum of tardiness of the jobs given a list schedule becomes NP-hard [NS03]. However, suppose we remove the objective function (*i.e.*, we only search for a schedule that satisfies the cumulative constraint without

(a) There are 3 machines, and we are given a list of jobs for each machine

(b) Each job is scheduled as soon as possible with respect to the cumulative constraint. First jobs j_1, j_4 and j_7 are scheduled. Then we try to schedule job j_8 at time 1 but two jobs (j_1 and j_4) already consume the resource, so the start time of j_8 is delayed to time 3 (when job j_1 ends). The other jobs are scheduled following the same rules.

Figure 3.3: Example of list-schedule with classical cumulative constraint. Pink jobs consume one unit of the cumulative resource, whereas white ones do not consume the cumulative resource. The limit of the resource is equal to 2.

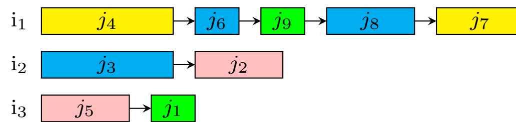


Figure 3.4: Example of list-schedule with group cumulative constraint. Whatever the start-times of jobs, this list-schedule cannot lead to a schedule which respects the GC constraint.

minimizing the tardiness sum). In that case, there always exist at least one solution with the same order as the one defined in the list schedule, and we can build one of these solutions greedily, by considering jobs in the order of the list l_i for each machine $i \in M$, and scheduling each job as soon as possible with respect to cumulative constraints.

For example, let us consider the list schedule displayed in Fig. 3.3a, and let us assume that pink jobs require one unit of resource (whereas white ones do not require any resource), and the capacity of this resource is 2. In this case, the greedy approach computes start times displayed in Fig. 3.3b.

3.3.2 Group cumulative constraint

However, this is no longer true for the GCSP and some list schedules may not be consistent, *i.e.*, there does not exist a solution with the same order of jobs even when there is no objective function to optimize.

For example, let us consider the list schedule displayed in Fig. 3.4 (which has the same jobs as in Fig. 3.1). We cannot find start times that satisfy the group cumulative constraint for this list-schedule when $l = 2$. Indeed, because of the sequence of jobs on machine i_1, when job j_9 starts, jobs j_4 and j_6 are already started, and jobs j_8 and j_7 are not yet ended. Thus when job j_9 starts, the yellow and the blue groups are active. So the yellow, blue, and green groups are necessarily active simultaneously at a given time. Hence this list-schedule cannot lead to a schedule that respects the GC constraint.

However, deciding whether a list schedule is consistent or not is difficult in the general case.
More precisely, let us denote *LS-GCSP* the problem of deciding whether there exists a solution of the *GCSP* that is consistent with a given list schedule, where a list schedule is consistent with a solution of *GCSP* if and only if, for every $j_1, j_2 \in \mathcal{J}$ such that j_1 occurs before j_2 in the same list, we have $C_{j_1} \leq B_{j_2}$.

Theorem 3.3.1. *LS-GCSP* is $\mathcal{NP}$-complete.

Proof. *LS-GCSP* clearly belongs to $\mathcal{NP}$ as we can check in polynomial time if a given assignment is a solution of GCSP, which is consistent with a list schedule.

Now, let us show that *LS-GCSP* is $\mathcal{NP}$-complete by reducing the Pathwidth problem to it.

Definition 3.3.1 (Pathwidth Problem)**.**
Given a connected graph $G = (\mathcal{N}, \mathcal{E})$ (such that $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ a set of edges) and an integer w, Pathwidth aims at deciding whether there exists a sequence $(\mathcal{N}_1, ..., \mathcal{N}_n)$ of subsets of $\mathcal{N}$ such that:

1. $\mathcal{N} = \bigcup_{i=1}^{n} \mathcal{N}_i$;
2. $\forall \{u, v\} \in \mathcal{E}, \exists i \in [1, n], \{u, v\} \subseteq \mathcal{N}_i$;
3. $\forall i, j, k \in [1, n], i \leq j \leq k \Rightarrow \mathcal{N}_i \cap \mathcal{N}_k \subseteq \mathcal{N}_j$;
4. $\forall i \in [1, n],\ \#\mathcal{N}_i \leq w$.

Pathwidth is $\mathcal{NP}$-complete [KAS79]. Fig 3.5a shows an example of an instance of the Pathwidth problem, and Fig 3.5b a solution for this instance.

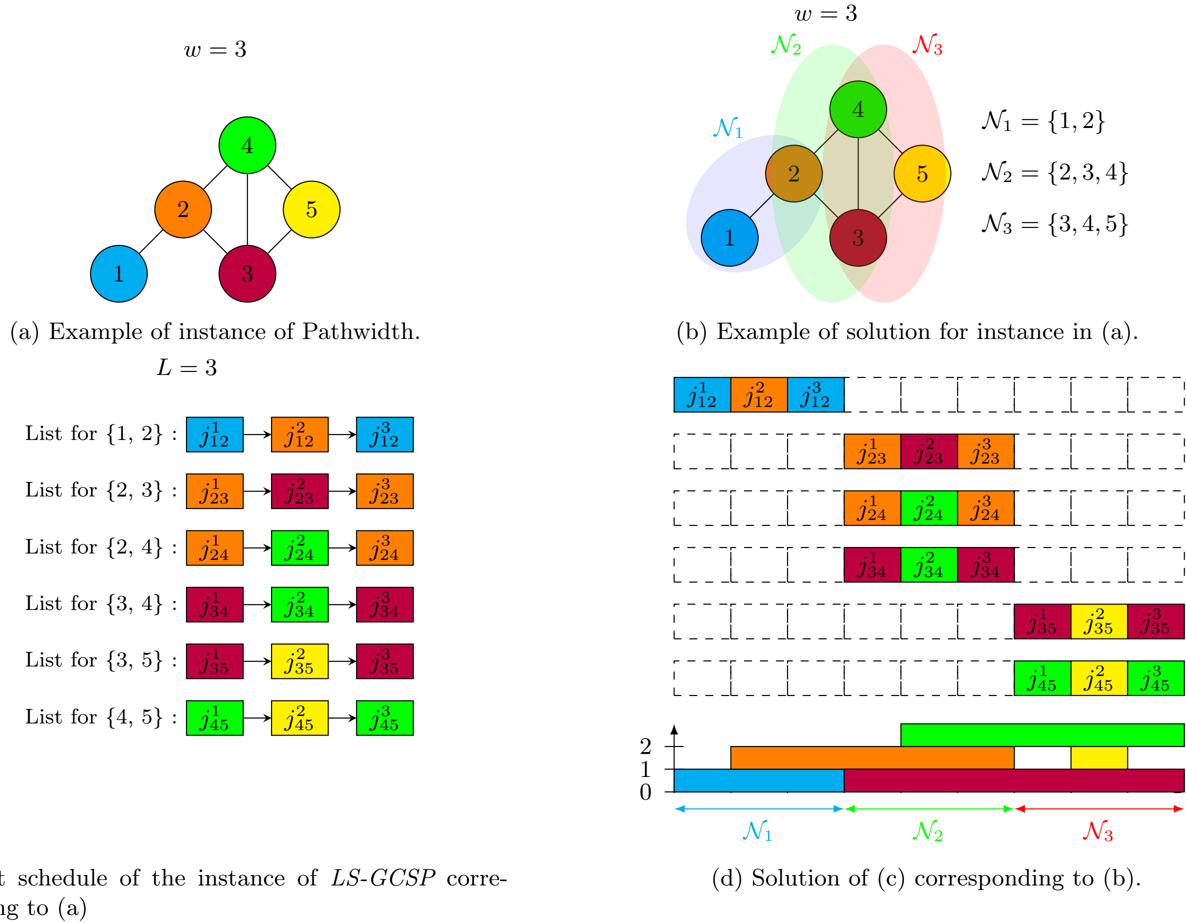
Reduction from Pathwidth to LS-GCSP: Let us first show how to construct an instance of *LS-GCSP* given an instance of Pathwidth defined by a graph $G = (\mathcal{N}, \mathcal{E})$ and an integer w. We assume that nodes of $\mathcal{N}$ are numbered from 1 to $|\mathcal{N}|$. For each edge $\{u, v\} \in \mathcal{E}$, we define three jobs denoted j^1_{uv}, j^2_{uv}, and j^3_{uv} such that every job has a processing time equal to 1. The partition $\mathcal{P}$ associates one group g_u with every vertex u such that

$$g_u = \{j^1_{uv}, j^3_{uv} : \{u, v\} \in \mathcal{E} \wedge u < v\} \cup \{j^2_{uv} : \{u, v\} \in \mathcal{E} \wedge u > v\}$$

In other words, for each edge $\{u, v\} \in \mathcal{E}$ such that $u < v$, j^1_{uv} and j^3_{uv} belong to group g_u whereas j^2_{uv} belongs to group g_v.
There are $|\mathcal{E}|$ machines, and the list schedule associates the list $(j^1_{uv}, j^2_{uv}, j^3_{uv})$ with every edge $\{u, v\} \in \mathcal{E}$ such that $u < v$. Finally, we set the limit l to w.
Fig. 3.5c gives an example of this transformation. Clearly, this transformation is polynomial with respect to the size of the Pathwidth instance as J contains $3 * |\mathcal{E}|$ jobs.

'Yes' answer kept from Pathwidth to *LS-GCSP*: Now, let us show that every solution $(\mathcal{N}_1, ..., \mathcal{N}_n)$ of an instance of Pathwidth corresponds to a solution of the corresponding instance of *LS-GCSP*. To this aim, we show how to define the start time $B_{j^k_{uv}}$ of every job j^k_{uv} associated with an edge $\{u, v\} \in \mathcal{E}$, with $k \in \{1, 2, 3\}$:
First, let a be the index of the first subset in $(\mathcal{N}_1, ..., \mathcal{N}_n)$ which contains both u and v (*i.e.*, $a = \min\{b \in [1, n] : \{u, v\} \subseteq \mathcal{N}_b\}$);
Then we define $B_{j^1_{uv}} = 3 * a - 3$, $B_{j^2_{uv}} = 3 * a - 2$, and $B_{j^3_{uv}} = 3 * a - 1$; end times are computed by adding the processing time 1 to every start time.

In Fig. 3.5d, we display start and end times computed for a solution of the Pathwidth instance of Fig. 3.5a. We can easily check that start and end times are consistent with the list schedule $(3 * a - 3 < 3 * a - 2 < 3 * a - 1)$.
To show that start and end times satisfy *GCSP*, we have to show that the number of active groups never exceeds l. If we consider a time t with $3 * a - 3 \leq t \leq 3 * a - 1$ ($a \in [1, n]$), then the only groups that can be active at time t are those associated with nodes in $\mathcal{N}_a$. Indeed let g_u be a group such that g_u is active at t. As g_u is active at t, there exists v_1 and k_1 such that $B_{j^{k_1}_{uv_1}} \leq t$ (or $B_{j^{k_1}_{v_1u}} \leq t$) and v_2 and k_2 such that $B_{j^{k_2}_{uv_2}} > t$ (or $B_{j^{k_2}_{v_2u}} > t$). So there exists $a_1 \leq a$ such that $B_{j^{k_1}_{uv_1}} = 3 * a_1 - 3$ or $B_{j^{k_1}_{uv_1}} = 3 * a_1 - 2$ or $B_{j^{k_1}_{uv_1}} = 3 * a_1 - 1$. And so $\{u, v_1\} \in \mathcal{N}_{a_1}$. A similar reasoning allows us to conclude that there exists $a_2 \geq a$ such that $\{u, v_2\} \in \mathcal{N}_{a_2}$. Hence we have $a_1 \leq a \leq a_2$ and $u \in \mathcal{N}_{a_1}$ and $u \in \mathcal{N}_{a_2}$.



(a) Example of instance of Pathwidth.

(b) Example of solution for instance in (a).

(c) List schedule of the instance of *LS-GCSP* corresponding to (a)

(d) Solution of (c) corresponding to (b).

Figure 3.5: Transformation of an instance of the Pathwidth problem into an instance of the *LS-GCSP*

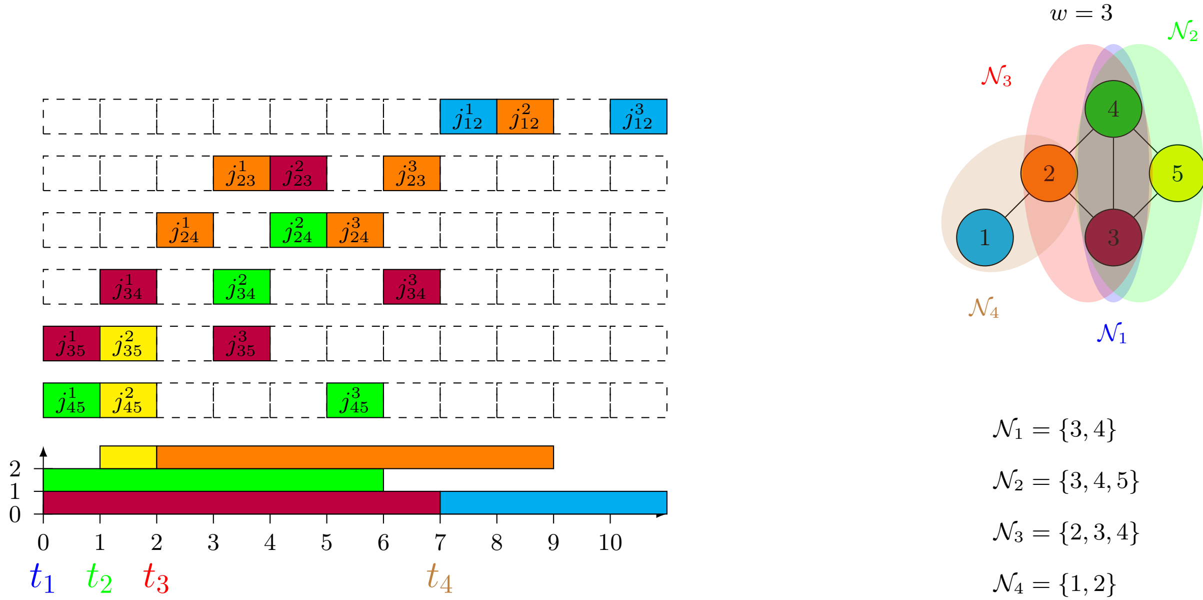


Figure 3.6: Transformation of a solution of an instance of the *LS-GCSP* into a solution of the *Pathwidth* problem. At time $t_1 = 0$ groups 3 (red) and 4 (green) become active, so $\mathcal{N}_1 = \{3, 4\}$. At time $t_2 = 1$ group 5 (yellow) becomes active and group 3 and 4 are still active so $\mathcal{N}_2 = \{3, 4, 5\}$. At time $t_3 = 2$ group 2 (orange) becomes active, and group 5 is not active anymore, so $\mathcal{N}_3 = \{2, 3, 4\}$. Finally at time t_4, group 1 (blue) becomes active and group 2 is the only other group which is still active, so $\mathcal{N}_4 = \{1, 2\}$.

As $(\mathcal{N}_1, ..., \mathcal{N}_n)$ is a solution of the instance of Pathwidth, and because of the point 3 in the definition of the Pathwidth problem, we know that $u \in \mathcal{N}_a$.
Hence the only groups that can be active at time t are those associated with nodes in $\mathcal{N}_a$. Thus, the number of active groups at t is lower than $|\mathcal{N}_a|$, which is lower than w because of the point 4 in the definition of the Pathwidth problem. Hence the number of active groups at t is lower than l (because $l = w$).
Thus the defined start and end times correspond to a solution of the instance of *LS-GCSP*

'Yes' answer kept from *LS-GCSP* to Pathwidth: Finally, let us show that every solution of the instance of *LS-GCSP* built from an instance of Pathwidth corresponds to a solution of this Pathwidth instance. A solution of an instance of *LS-GCSP* is an assignment of values to B_j and C_j for every job $j \in \mathcal{J}$ (defining start and end times of j). For each node $u \in \mathcal{N}$, we have a group of jobs g_u, and the start time B_u of this group is the smallest start time of its jobs (*i.e.*, $B_u = \min\{B_j : j \in g_u\}$) whereas the completion time C_u of this group is the largest end time of its jobs (*i.e.*, $C_u = \max\{C_j : j \in g_u\}$). Let $\mathcal{T} = \{B_u : u \in \mathcal{N}\}$ be the set of all group start times, and let $(t_1, \ldots, t_{|\mathcal{T}|})$ be the ordered sequence of values in $\mathcal{T}$. The solution of the Pathwidth instance is $(\mathcal{N}_1, \ldots, \mathcal{N}_{\#\mathcal{T}})$ such that for each $i \in [1, |\mathcal{T}|]$, $\mathcal{N}_i = \{u \in \mathcal{N} : B_u \leq t_i < C_u\}$.
An example of such a construction of a Pathwidth solution from a *LS-GC* solution is given in Fig 3.6.
We can check that $(\mathcal{N}_1, \ldots, \mathcal{N}_{\#\mathcal{T}})$ is a solution of the Pathwidth instance. We will do it point by point according to the definition 3.3.1

1. it is trivial to verify that $\mathcal{N} = \bigcup_{i=1}^{n} \mathcal{N}_i$;
2. for each edge $\{u, v\} \in \mathcal{E}$ with $u < v$, the list $(j^1_{uv}, j^2_{uv}, j^3_{uv})$ ensures that when j^2_{uv} starts both $\mathcal{G}_u$ and $\mathcal{G}_v$ are active groups. Hence, for $i \in [1, |\mathcal{T}|]$ such that $t_i = max(B_u, B_v)$, $\{u, v\} \in \mathcal{N}_i$;

3. Let us take $i, j, k \in [1, |\mathcal{T}|]$ with $i \leq j \leq k$, $u \in \mathcal{N}_i$ and $u \in \mathcal{N}_k$ then $B_u \leq t_i \leq t_j \leq t_k < C_u$. Hence $u \in \mathcal{N}_j$, and so $\mathcal{N}_i \cap \mathcal{N}_k \subseteq \mathcal{N}_j$;
4. $\forall i \in [1, \#\mathcal{T}]$, all groups of $\mathcal{N}_i$ are active at time t_i. So $|\mathcal{N}_i| \leq l = w$

□

3.4 Discussion

In this chapter we introduce a new scheduling problem: the *Group Cumulative Scheduling Problem (GCSP)*. We study this problem from a theoretical point of view. In particular, we compare it with known problems and highlight common features as well as differences. We also study the computational complexity of the problem (according to the framework introduced in chapter 1). In particular, we show that some list schedules cannot be transformed into solutions, even when there is no objective function to optimize, and that the problem of deciding whether a list schedule can be transformed into a solution or not is $\mathcal{NP} - complete$.
Although this problem is different from the scheduling problem studied so far, it remains a scheduling problem. Hence the methods described so far to solve scheduling problem can still be used. However they need to be adapted. These adaptations are the topic of the next chapter.

Chapter 4

New approaches for solving the GCSP

Contents

4.1 Model for Constraint Programming . . . 73
4.2 Adapting solving methods to the *GCSP* . . . 74
4.2.1 Local search . . . 74
4.2.2 Greedy constructions . . . 76
4.2.3 Ant Colony Optimization . . . 80
4.3 New Hybrid CPO-ACO approach . . . 81
4.4 Discussion . . . 82

As described in the previous chapter, and especially because of theorem 3.3.1, solving methods described so far need to be adapted to the *GCSP*. In this chapter we describe methods to deal with the GCSP. More precisely in section 4.1 we give a *CP* model in order to solve the GCSP. Then, in section 4.2 we show how we can adapt known solving methods to solve *GCSP*. Finally in section 4.3 we present a new algorithm which combines *CP Optimizer* and *ACO*. This new algorithm can be used to solve the GCSP as well as other scheduling problems presented in chapter 1.

4.1 Model for Constraint Programming

In order to ease the modeling of the *GCSP* variants, we introduce the *GC* constraint:

Definition 4.1.1 (*Group Cumulative* constraint)**.**
Given a set $\mathcal{J}$ of jobs, a partition $\mathcal{P}$ of $\mathcal{J}$ in $|\mathcal{P}|$ groups (such that each job $j \in \mathcal{J}$ belongs to exactly one group $g \in \mathcal{P}$), an integer limit l and, for each job $j \in \mathcal{J}$, an integer variable B_j (resp. C_j) corresponding to the start time (resp. end time) of j, the constraint $GC_{\mathcal{J},\mathcal{P},l}(\{B_j : j \in \mathcal{J}\}, \{C_j : j \in \mathcal{J}\})$ is satisfied if and only if $|\{g \in \mathcal{P} : \min_{j\in g} B_j \leq t < \max_{j\in g} C_j\}| \leq l$ for any time t.

We can easily decompose GC using a classical cumulative constraint. To this aim, we associate a new interval variable F_g with every group $g \in \mathcal{P}$. This variable corresponds to a fictive job, which starts with the group's earliest job and ends with its latest job and consumes one resource unit. A simple cumulative constraint on these fictive jobs ensures that the number of active groups never exceeds l.

More precisely, Fig. **??** describes a CPO model of this decomposition:

- Constraint (4.1) ensures that, for every group g, the fictive job variable F_g spans over all jobs in the group;
- Constraint (4.2) defines the cumul function (denoted *Active*) corresponding to the case where each fictive job consumes one unit of the resource;

$$span(F_g, \{a_j : j \in g\}) \qquad \forall g \in \mathcal{P} \tag{4.1}$$

$$Active = \sum_{g \in \mathcal{P}} pulse(F_g, 1) \tag{4.2}$$

$$lowerOrEqual(Active, l) \tag{4.3}$$

- Constraint (4.3) ensures that *Active* never exceeds l, thus ensuring the cumulative constraint on fictive jobs.

This model uses the CPO language. However, it can be easily modeled in any Constraint Programming language which uses scheduling variables and constraints (interval variables, sequence variables, cumulative constraints, ...). In particular the *span* constraint can be replaced by the two following constraints:

$$startOf(F_g) = \min_{j \in g} startOf(a_j) \qquad \forall g \in \mathcal{P} \tag{4.4}$$

$$endOf(F_g) = \max_{j \in g} endOf(a_j) \qquad \forall g \in \mathcal{P} \tag{4.5}$$

Then the cumulative constraint is applied on the F_g intervals.

If we can quite easily decompose the GC constraint with classical CP constraints, we shall see in experimental results reported in Chapters 5 and 6 (see Fig. 5.2, for example) that adding this constraint to a scheduling problem degrades CPO performance. This may be explained by the fact that CPO exploits precedence relations to solve scheduling problems [LRSV18]: all temporal constraints are aggregated in a temporal network whose nodes represent interval start and end time-points and whose arcs represent precedence relations. Also, CPO integrates a *Large Neighborhood Search (LNS)* component based on the initial generation of a directed graph whose nodes are interval variables and edges are precedence relations between interval variables. However, when using the *GC* constraint, such precedence relations are harder to exploit because of theorem 3.3.1.

It is important to notice that several methods exist to deal with cumulative resources (timetable techniques, edge-finding, *not first not last*, energy precedence) [Bon17]. Some of these methods derive from the notion of *energy*. Given a job $j \in \mathcal{J}$ its energy is defined as the product of its duration times its resource consumption: $p_j \times \lambda_j$. In the case of the *GCSP*, the group durations is unknown beforehand. As aforementioned, approximating a group's duration with precision is difficult and the methods which use energy to solve cumulative scheduling problems may not be efficient when applying them to the *GCSP*.

4.2 Adapting solving methods to the *GCSP*

4.2.1 Local search

Local search is harder to implement when using the *GC* constraint. Indeed, most of the neighborhoods used for scheduling problems deal with sequences of jobs on the different machines (list-schedules). Such neighborhoods are especially efficient when it is easy to compute a schedule from a list-schedule. However theorem 3.3.1 shows that, for the *GCSP*, such computation is *NP-Complete*.
Furthermore, as explained in section 2.3.3, tabu search is more efficient if we can evaluate the neighborhoods incrementally. However, for the *GCSP*, such incremental computation is more complicated. Indeed, let us assume we have an instance with three machines, if we swap two jobs on machine i_3 for example, for the *GCSP* (and also with classical cumulative constraints), this move can have an impact on jobs that are on machines i_1 and i_2.

Algorithm 4: Algorithm to obtain a schedule from a list-schedule for the *GCSP*

Input : A *GCSP* $(\mathcal{M}, \mathcal{J}, \mathcal{P}, l)$ and a list-schedule
Output: A feasible schedule or failure

```
1  Schedule each job as soon as possible according to the list-schedule
2  Let G_moved be an empty sequence
3  while The GC constraint is not satisfied do
4      if There exist two sequences x and s s.t. G_moved = x.s.s then
5          return failure
6      Let t_min be the smallest t such that |{g ∈ P : B_g ≤ t < C_g}| > l
7      G_active ← {g ∈ P | B_g ≤ t_min < C_g}
8      conflictSolved ← False
9      G' ← G_active
10     while not conflictSolved do
11         g' ← argmax_{g' ∈ G'} B_g'
12         G" ← {g" ∈ G_active \ {g'} | ∀j" ∈ g" ∀j' ∈ g', I_j" ≠ I_j' ∨ B_j" < B_j'}
13         if G" ≠ ∅ then
14             g" ← argmin_{g" ∈ G"} C_g"
15             Introduce idle times before jobs of g' s.t. ∀j' ∈ g' B_j' ≥ C_g"
16             conflictSolved ← true
17             Add the pair (g', g") at the end of G_moved
18         else
19             Remove g' from G'
20             if G' = ∅ then
21                 return failure
22 return The current schedule
```

In order to deal with the *GCSP*, one must define a procedure that can compute the start and end times of jobs that respect both the list-schedule and the *GC* constraint (and which tries to minimize the objective function). Such procedure is necessarily approximated if we want to evaluate neighborhoods in polynomial time (unless $\mathcal{P} = \mathcal{NP}$) because of theorem 3.3.1. Algo. 4 gives such a procedure, and Fig. 4.1 illustrates it on a small example. It works as follow:

- First it computes start and end times of jobs as if there is no *GC* constraint (line 1)
- Then, while the *GC* is not respected, it looks for the first point in time t where the constraint is violated
- Then we compute the set G_{active} of groups which are active at that time (i.e., which participate at the violation of the constraint, line 7)
- Then we try to solve the conflict at hand. To do so, we select two groups g' and g" in G_{active} and we introduce idle times such that g' starts after the end of g". Doing so, g' and g" will not be active simultaneously, and hence the number of active groups at t will decrease of at least one unit.
- To do so we select for g' the group which starts the latest, line 11.
- Then we have to select g". However not all the active groups can be considered for g". Indeed we want to introduce idle times on machines so that g' starts after the end of g". To do so we increase the start times of the jobs of g' so that $\forall j' \in g'$ $B_{j'} \geq C_{g"}$ (line 15). However, when we increase the start time of a job $j' \in g'$, we also increase the start and end times of all the jobs that are on the same machine as j' and which are scheduled after j'. In

particular if there is a job j" of g", which is after j' on the same machine, then as we increase the start time of j' we also increase the end time of j"; and, hence as we increase $B_{j'}$ we also increase $C_{g"}$, and thus, in such a case, it is not possible to have $B_{j'} \geq C_{g"}$. To avoid this situation, we limit the set of candidates G" (line 12) to the group g" such that for each job $j" \in g"$ and for each job $j' \in g'$ either j" and j' are on different machines or j' starts after j".

- If G" is not empty, then, in lines 14 and 15 we select the group in G" which ends the soonest (in order to minimize the length of idle times), and we introduce idle times such that g' starts after g". Doing so we have solved the conflict at time t_{min}.
- If G" is empty, then we cannot moved g' such that it starts after another group of G_{active}. In such a case we remove g' from G' and we start with another group. If $G' = \emptyset$, then we have failed to solve this conflict, and we return failure.
- It is important to notice that we can have an instance in which we move group g_1 after group g_2, and in the next iteration we move g_2 after g_3, and then g_3 after g_1, and once again g_1 after g_2, and we repeat this cycle indefinitely (see Fig. 4.2). In order to avoid this situation, we maintain a sequence G_{moved} which contains all the couples of groups (g', g") such that we have made g' start after g" (line 17). As soon as a sub-sequence s appears twice consecutively in G_{moved}, we stop the search and return failure (line 4). More formally, if there exists two sub-sequences of job couples x and s such that $G_{moved} = x.s.s$, we return failure.

It is important to notice that, as stated in theorem 3.3.1, computing a schedule from a list-schedule for the *GCSP* is *NP-Complete*. In particular, Algo. 4 sometimes returns "failure" whereas there does exist a schedule that respects both the list-schedule given as input and the *GC* constraint.
Further remarks must be made about G_{moved}:

- Different strategies can be implemented in order to stop the search when it seems that the algorithm is stuck. For example, we could stop the search as soon as we add another time a couple $(g', g")$ which is already in G_{moved}. Such a strategy stops the search earlier than our strategy. Indeed, as illustrated in Fig. 4.1, where (B, R) appears twice in G_{moved}, such a strategy sometimes returns failures whereas with some additional iterations a solution can be found.
- As the number of job couples is finite, we cannot build an infinite sequence such that for all sub-sequences of job couples x and s $G_{moved} \neq x.s.s$. Hence this prove that the algorithm terminates.
- However, the complexity of the algorithm is exponential with respect to the number of groups in the instance in the worst case. Indeed, we possibly need to add an exponential number of job couples to G_{moved} before having two sub-sequences of job couples x and s such that $G_{moved} \neq x.s.s$.
- In practice, the algorithm often terminates within a reasonable number of iterations.

4.2.2 Greedy constructions

As seen in Section 2.1, solutions of (parallel machines) scheduling problems are often computed iterating the two following steps:

1. Select the machine which ends the sooner (break ties arbitrary)
2. Select a job among the unscheduled ones and add it at the end of the selected machine (and compute its start and end times)

until all jobs are assigned.
For the *GCSP*, if one can select any job among the unscheduled ones, it is possible to get sequences of jobs on machines such that no schedule which respects the *GC* constraint can be computed.

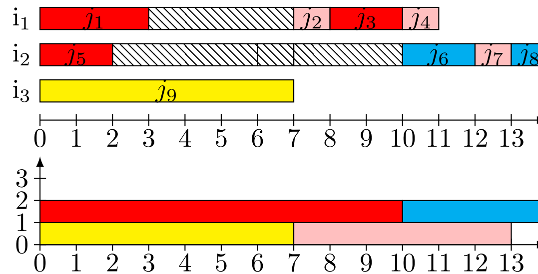
(a) At the start of the algorithm we have $G_{moved} = \emptyset$. The first conflict occurs at $t = 2$. At that time $\mathcal{G}_{active} = \{R, B, Y\}$. We take $g' = B$ as the blue group is the one which starts the latest. We have $\mathcal{G}" = \{R, Y\}$ and we take $g" = R$ as the red group is the one which ends the soonest. So we put B after R by introducing idle time before j_6 so that it starts after the end of j_3.

(b) Now, $G_{moved} = \{(B, R)\}$. A second conflict occurs at $t = 3$. At that time $\mathcal{G}_{active} = \{R, P, Y\}$. We take $g' = P$. We have $\mathcal{G}" = \{Y\}$ ($R \notin \mathcal{G}"$ because we cannot make P start after the end of R, as j_3 is on the same machine as j_2 and starts after j_2). So we put P after Y. We introduce idle time before j_2 so that it starts after the end of j_9.

(c) Now, $G_{moved} = \{(B, R), (P, Y)\}$. A third conflict occurs at $t = 6$. At that time $\mathcal{G}_{active} = \{R, B, Y\}$. We take $g' = B$, and we have $\mathcal{G}" = \{Y, R\}$, and thus, $g" = Y$. Hence we introduce idle time before j_6 so that it starts after the end of j_9.

(d) $G_{moved} = \{(B, R), (P, Y), (B, Y)\}$. A fourth conflict occurs at $t = 7$. At that time $\mathcal{G}_{active} = \{R, B, P\}$. We take $g' = P$. However we have $\mathcal{G}" = \emptyset$ (j_2 can start after j_3 so $R \notin \mathcal{G}"$ and j_7 cannot start after j_8 so $B \notin \mathcal{G}"$). Hence we remove P from $\mathcal{G}'$ and we take $g' = B$. Now, we have $\mathcal{G}" = \{R\}$. Hence we introduce idle time before j_6 so that it starts after the end of j_3.

Figure 4.1

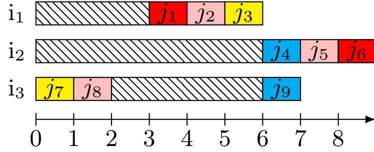


(e) Now, we have $G_{moved} = \{(B, R), (P, Y), (B, Y), (B, R)\}$, and there are no more conflicts. We can notice that (B, R) appears twice in G_{moved}.

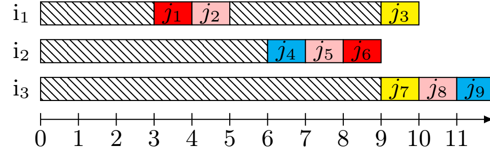
Figure 4.1: An example of execution of Algo. 4. There are three machines, four groups (yellow, red, blue, and pink, denoted respectively Y, R, B and P), and the limit l on the number of active groups at any time is equal to 2.

(a) At $t = 0$, $\mathcal{G}_{active} = \{R, B, Y\}$. We take $g' = R, g" = B$ and we add (R, B) to G_{moved}.

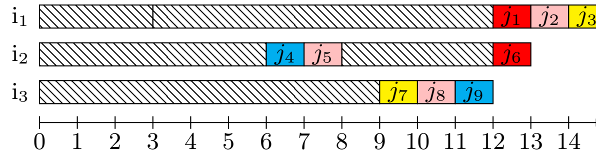
(b) At $t = 1$, $\mathcal{G}_{active} = \{B, P, Y\}$. We take $g' = P$. However, because j_3 is after j_2 and j_9 is after j_8, we have $\mathcal{G}" = \emptyset$. So we take $g' = B$, and $g" = Y$. We introduce idle times and we add (B, Y) to G_{moved}



(c) At $t = 1$, $\mathcal{G}_{active} = \{R, P, Y\}$. Taking $g' = R$ leads to $\mathcal{G}" = \emptyset$, and the same happens with $g' = P$. So, we take $g' = Y$, and we have $g" = R$. We introduce idle times and we add (Y, R) to G_{moved}



(d) At $t = 6$, $\mathcal{G}_{active} = \{R, P, B\}$. Taking $g' = B$ or $g' = P$ leads to $\mathcal{G}" = \emptyset$. So, we take $g' = R$, and we have $g" = B$. We introduce idle times and we add (R, B) to G_{moved}

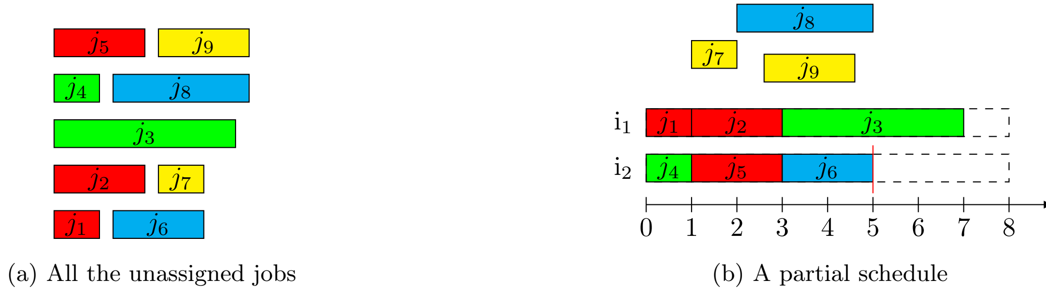


(e) At $t = 9$, $\mathcal{G}_{active} = \{B, P, Y\}$. Taking $g' = Y$ or $g' = P$ leads to $\mathcal{G}" = \emptyset$. So, we take $g' = B$, and we have $g" = Y$. We introduce idle times and we add (B, Y) to G_{moved}. After this move, we have $G_{moved} = ((R, B), (B, Y), (Y, R), (R, B), (B, Y))$. Hence we notice that, without a stopping criterium on G_{moved}, we indefinitely repeat the sequence $((R, B), (B, Y), (Y, R))$.

Figure 4.2: An example of execution of Algo. 4 where no solution can be found. There are three machines, four groups (yellow, red, blue, and pink, denoted respectively Y, R, B and P), and the limit l on the number of active groups at any time is equal to 2.

Algorithm 5: Construction of a solution for a *GCSP*

Input : A *GCSP* $(\mathcal{M}, \mathcal{J}, \mathcal{P}, l)$
Output: A schedule
1 $S \leftarrow \emptyset$
2 **while** $\mathcal{J} \neq \emptyset$ **do**
3 $\quad i_{next} \leftarrow \text{argmin}_{i \in \mathcal{M}}\ endTime(i)$
4 $\quad G_{open} = \{g \in \mathcal{P} | \exists j \in \mathcal{J}, g_j = g \text{ and } \exists j' \in S, g_{j'} = g\}$
5 $\quad$ **if** $|G_{open}| < l$ **then** $Cand \leftarrow \mathcal{J}$;
6 $\quad$ **else** $Cand \leftarrow \{j \in \mathcal{J} : g_j \in G_{open}\}$;
7 $\quad$ choose $j \in Cand$ which maximizes the ATCS formula (see sec. 2.1)
8 $\quad i_j \leftarrow i_{next}$
9 $\quad$ compute the start time B_j and the completion time C_j of j
10 $\quad$ remove j from $\mathcal{J}$ and add it to S



(a) All the unassigned jobs (b) A partial schedule

Figure 4.3: Example of *active*, *closed* and *open* groups. Red and green groups are closed because all their jobs are scheduled. The yellow group is also closed because none of its jobs is scheduled. The blue group is open because its job j_6 is scheduled whereas its job j_8 is not scheduled. If we consider Algo. 5, the next job must be assigned on machine i_2, at time 5. We can notice that, at that time, the green group is active. Hence, at $t = 5$ the green group is closed and active. Let's assume that the limit l of the *GCSP* is equal to 2. As $|G_{open}| < l$, any job can be scheduled. In particular, it is allowed to schedule job j_9 on i_2 at time $t = 5$. If we do so, as j_8 will necessary ends after $t = 6$ the blue, yellow, and green groups will be active at time $t = 6$, which violates the constraint. Hence, to avoid this situation, if we choose to schedule j_9 on i_2, we introduce one unit of idle time to ensure that the yellow group starts after the end of the green one.

Hence, to deal with the *GCSP*, one needs to adapt the ATCS procedure. Algo. 5 describes this new algorithm.
It is quite similar to the ATCS procedure, but it sometimes reduces the set of candidate jobs. To this aim, we first compute the set G_{open} of groups g such that at least one job of g has been scheduled in S and at least one job of g has not yet been scheduled. It is important to notice that the notion of *open* groups is different from the notion of *active* groups as illustrated in Fig 4.3. On the first hand, *open* groups are defined only on partial assignments, when some jobs of the group are scheduled whereas some others are not yet scheduled. In particular, the notion of *open* groups does not depend on the time t we consider. A group which is not *open* is said to be *closed*. On the other hand, *active* groups are defined for complete assignments and depend on the time t we consider. In order to ease the comprehension we extend the notion of *active* groups to partial assignments. In such a case, if a group g is closed, and none of its jobs are assigned we say that g is not active for any time $t \in [0, h]$. If g is closed and all its jobs are assigned we say that g is active at a time t if some of its jobs are started and some others are not yet ended at t (like in the case of complete assignments). Finally for an *open* group g we say that g is active at time t if $t \geq \min_{\substack{j \in g \\ j \text{ is scheduled}}} B_j$.

If $|G_{open}| < l$, then we can select any job of J (line 5). Indeed in such a case, we know that we can obtain a schedule

satisfying the GC by introducing idle times if necessary. Otherwise, $|G_{open}|$ is equal to l, and in this case, we restrict $Cand$ to the jobs that are not yet scheduled and that belong to an order of G_{open} (line 6). Note that this filtering procedure may remove from $Cand$ some jobs that could lead to better schedules. However, as it is NP-complete to decide if a job can be scheduled without violating the GC constraint, we use this simple filtering procedure to ensure feasibility. In line 9, sometimes we must introduce some idle time on machine i_{next}. Indeed it is possible that at time $C_{i_{next}}$ (the end time of machine i_{next}) the number of active groups is equal to l (see Fig. 4.3 for example). Hence we introduce idle times so that the start time of j is the smallest t such that the number of active groups at t is lower than l. Such a t necessary exists. Indeed, we know that at time $t = \max_{i \in \mathcal{M}} C_i$, the *active* groups are only the *open* ones. Hence at that time, because of the choice of $Cand$, either this number is lower than l or j belongs to an *open* group and we can schedule j without increasing the number of active groups.

4.2.3 Ant Colony Optimization

Classical ACO algorithms for scheduling problems (described in section 2.2) may be adapted in a straightforward way to the GCSP by reducing the set $Cand$ used at line 13 of Algo. 2 as done in lines 5-6 of Algo. 5. The two classical pheromone structures *Position* and *Jobs* recalled in Section 2.2 may be used for the GCSP.

However, when dealing with a GC constraint (or a classical cumulative constraint), it can be necessary to introduce some idle times on the machines to satisfy it. Hence these new characteristics for the problem should be taken into account. For this reason, we introduce a new pheromone structure, which consists in learning when each job should start. More formally, let $\mathcal{H}_{stepSize}$ be the time horizon, discretized according to a given step size *stepSize* (*i.e.*, $\mathcal{H}_{stepSize}$ is a finite set of contiguous time intervals such that the first interval starts at the beginning of the day, the last interval ends at the end of the day, and the size of each interval is equal to *stepSize*). For each couple $(j, \nu) \in \mathcal{J} \times \mathcal{H}_{stepSize}$, we define a pheromone trail $\tau(j, \nu)$ which represents the learned desirability of scheduling job j at time step ν. The pheromone factor used to compute the probability of selecting a job in Algo. 2 is defined by: $f_\tau(j) = \tau(j, \frac{C_i}{stepSize})$ where i is the considered machine, and C_i is the end time of machine i. In a similar way, given a schedule that we want to reward, in which a given job $j \in \mathcal{J}$ starts at B_j we increase the pheromone trail $\tau\left(j, \left\lfloor \frac{B_j}{stepSize} \right\rfloor\right)$ and we decrease all pheromone trails $\tau(j, h')$ for $h' \in \mathcal{H}$.

It is important to notice that with this structure, we must memorize the pheromone value for each pair $(j, \nu) \in \mathcal{J} \times \mathcal{H}_{stepSize}$, which represents $n * \frac{h}{stepSize}$ values. Furthermore, the horizon h is often a huge value. Hence the pheromone can take a huge amount of memory. However, in the *MMAS* framework, each pheromone trail is initialized with value τ_{max}, and a trail which is never rewarded decreases of $(1 - \rho)$ at each iteration, until reaching value τ_{min}. Hence, a never rewarded trail has value $\max((1 - \rho)^k \tau_{max}, \tau_{min})$ at iteration k. Hence to save time and memory, we proceed as follows:

- For each job $j \in \mathcal{J}$ we maintain an interval $[s^j_{min}, e^j_{max}]$. At the beginning of the search the interval is empty. The idea is to keep in memory only the values of the pheromone trails (j, ν) for $\nu \in [s^j_{min}, e^j_{max}]$.
- Whenever a pheromone trail (j, ν) must be rewarded, if $\nu \in [s^j_{min}, e^j_{max}]$, then it is rewarded in classical way; otherwise we extend the interval such that $\nu \in [s^j_{min}, e^j_{max}]$ (*i.e.*, if $\nu > e^j_{max}$ (resp. $\nu < s^j_{min}$), then we set $e^j_{max} = \nu$ (resp. $s^j_{min} = \nu$)).
- Evaporation is only applied in interval $[s^j_{min}, e^j_{max}]$.
- Whenever one wants to know the value of $\tau(j, \nu)$ for $(j, \nu) \in \mathcal{J} \times \mathcal{H}_{stepSize}$, if $\nu \in [s^j_{min}, e^j_{max}]$, then we return its value (which is maintained), otherwise we return $\max((1 - \rho)^k \tau_{max}, \tau_{min})$, where k is the current iteration.

Algorithm 6: Hybrid approach combining CPO and ACO

Input : A scheduling problem P with jobs $\mathcal{J}$ and machines $\mathcal{M}$
Parameters α, β, n_{ants}, pheromone structure and pheromone system, τ_{min}, τ_{max}
Parameters k, b and g
Output: A schedule, *i.e.*, the machine I_j, start time B_j and completion time C_j for each job $j \in \mathcal{J}$

```
1  iter ← 0
2  Initialize pheromone trails to τ_max
3  while Stopping criterion not reached do
4      sol ← greedilyBuildSchedules(P, α, β, n_ants) (defined in Algo. 2)
5      update pheromone trails according to sol
6      iter ← iter + 1
7      if iter = 0 mod (k) then
8          Execute CPO using the best solution found in the last k iterations as starting point until b backtracks
           have been done
9          Update pheromone trails with the solution found by CPO
10         b ← b * g
11 return the best constructed solution
```

4.3 New Hybrid CPO-ACO approach

Many hybrid approaches combine exhaustive solvers (such as CP or Integer Linear Programming, for example) with meta-heuristics [BPRR11]. Some of these hybrid approaches are referred to as *matheuristics* [MSV10]. A well-known example of hybrid approach is LNS [Sha98] which uses an exact approach such as CP to explore the neighborhood of a local search.

Different hybrid CP-ACO approaches have been proposed such as, for example, [ME04, Mey08, KAS08, KAS10, Sol10, DGRU13]. Some approaches use constraint propagation during the construction of solutions by ACO (lines 13-18 of Algo. 2), to filter the set of candidate components and remove those that do not satisfy constraints [Mey08, KAS08]. Some other approaches use ACO to learn ordering heuristics which are used by CP [ME04, KAS10]. In [DGRU13], a bi-level hybrid process is introduced where ACO is used to assign a subset of variables, and the remaining variables are assigned by CP.

In this section, we introduce a new hybrid CPO-ACO approach where ACO and CPO are alternatively executed and exchange solutions: solutions found by ACO are used as starting points for CPO, whereas solutions found by CPO are used to update pheromone trails. Algo. 6 describes it more precisely. The ACO part (lines 2 to 5) is similar to Algo. 2. Then, every k iterations (line 7), we call CPO. When calling CPO, we supply it with the best solution constructed during the k last iterations of ACO, and this solution is used by CPO as a starting point (line 8). Each call to CPO is limited to a given number of backtracks. Once CPO has reached this limit, we get the best solution found by CPO and update pheromone trails according to this solution (line 9).

The limit on the number of backtracks follows a geometric progression, as often done in classical restart strategies [Wal99]: the first limit is equal to b, and after each call to CPO, this limit is multiplied by g (line 10). Hence, our hybrid CPO-ACO approach may be viewed as a particular case of restarts where ACO is run before each restart to provide a new initial solution, and the best solution after each restart is given back to ACO to reinforce its pheromone trails.

Our hybrid CPO-ACO algorithm has three parameters along with the ACO parameters: the number k of ACO cycles, which are executed before each call to CPO, and the values b and g used to fix the limit on the number of backtracks of CPO.

4.4 Discussion

In this chapter we study the consequences of the theorem 3.3.1 on solving approaches. In particular, we show that it impacts the methods presented in chapter 2. In order to solve the *GCSP* with *Constraint Programming* we give a decomposition (using existing constraints) of the *GC* constraint. In particular we use the *span* and *cumulative* constraints. Another approach can consist in introducing a new global constraint with its own propagators in order to propagate the *GC*. This is let as future work.
For *ACO*, the main difference consists in only considering a subset of the set of candidates whenever a new job must be added to a solution under construction in order to ensure that the greedy constructions always lead to a valid solution.
For *tabu search* we develop a new algorithm (Algo. 4) in order to convert a list-schedule into a schedule (if it is possible). As this problem is $\mathcal{NP}-Complete$, this method is a heuristic one, *i.e.*, it may fail whereas a valid schedule exists.
Finally in section 4.3 we introduce a new algorithm which combines *CPO* and *ACO*. Using this algorithm, *CPO* and *ACO* takes turn, each exploiting the solution found by the other.

In order to evaluate the *GC* constraint on the algorithms from an experimental point of view, we compare the results of these algorithms on a set of instances. This experimental evaluation is the subject of chapter 6. Before commenting these results, we introduce our new benchmark in the next chapter. In particular, we give a short survey of scheduling applications in industry and we present our industrial context, and especially the Infologic company.

Part III

Experimental Evaluation on a New Benchmark

Chapter 5

Description of the Benchmark

Contents

5.1 Description of our applicative context **85**
5.1.1 Scheduling applications 85
5.1.2 ERP 86
5.1.3 Infologic and Copilote 86
5.2 Features of the data-set **87**
5.3 Discussion **89**

In this chapter we introduce our new benchmark. First, we give a short survey of scheduling applications in section 5.1. We also present our industrial context and, especially the Infologic company. In particular, we describe the *Copilote ERP* developed by Infologic and we describe a real industrial application of the GCSP that occurs in this context. We also explain the interest of studying different scheduling problems, with different constraints and objective functions for an ERP editor. In section 5.2 we describe more precisely our data-set.

5.1 Description of our applicative context

5.1.1 Scheduling applications

There are many applications of scheduling problems. For example, [BZ09] applies scheduling strategies in automotive R&D Projects, [Mol05] deals with an order picking problem.
More generally, [HMB+14] lists several fields in which scheduling models and algorithms have led to substantial benefits. Some fascinating figures for specific examples are given: the application of scheduling theories increased the factory output of 30% in the dairy industry, in a pulp and paper plant scheduling techniques allow an increase of 2% of the efficiency of the plants (with an estimated saving between 10-20 million EUR/year), in the crude-oil blend field this technique allows to save approximately 2.85 million USD/year. The article gives other examples of such success stories by applying scheduling theories.
[GM14] gives a survey about remanufacturing scheduling systems, both from a theoretical point of view and from an industrial point of view. Remanufacturing is a sustainable business practice that allows products no longer functional to enter the remanufacturing process to be refurbished or disassembled into usable modules, components, or materials. Their survey is made of data collected from the automotive parts remanufacturers Association (APRA), a major, established, professional association for automotive parts remanufacturers, which represents approximately 37% of all U.S. remanufacturing revenue. It gives interesting figures about the practice of scheduling (models used, objective functions) and the potential benefits, and the difficulties met to apply the theory. It also evaluates the gap between theoretical results and industrial results.
[Pin16], along with its great description of scheduling problems, lists several industrial softwares that help industrial

companies schedule their operations. It also emphasizes the difficulties linked to the deployment of scheduling tools. Indeed, although using scheduling tools can be a vested investment, many projects also exist where applying scheduling in industry fails. [SW96] lists several reasons which help to explain such failures. [Wie97] also explores this gap between theory and practice. A human scheduler's role is especially highlighted along with the human-computer interaction. It gives some insights into why some scheduling tools are used, whereas others are abandoned. In particular, the quality of the solutions found is not necessarily the main concern of users.

5.1.2 ERP

Enterprise resource planning (ERP) is the integrated management of main business processes, often in real-time and mediated by software and technology. It is used to plan and manage all the core supply chain, manufacturing, services, financial, and other organizational processes.
ERPs are made of several modules specialized in a particular part of the company business (for example, sales forecast or production planning). ERPs are often used in the replacement of different specific softwares. Such specialized softwares are sometimes more efficient on their task than a module of an ERP. However, the interface with other systems is often complicated, and maintaining the consistency of data in the whole system becomes a difficult challenge. With an ERP, this consistency is more comfortable to achieve, which increases efficiency and robustness. ERP has gained more and more interest since the end of the twentieth century. This growing interest is reflected both in the market weight of ERPs, and the number of academic publications [CFBA16].

From an editor's point of view, ERPs are developed to be used by many companies. Obviously, each company has its own constraints, and each business is different. Hence ERPs need to be highly customizable to be competitive. However, for ERPs with many modules, this customization can be challenging. Recently, some techniques [Cha18] have been developed to ease this customization.

Furthermore, some aspects of customization need expert knowledge on the problem at hand. It is especially the case for the production-scheduling part of ERPs. As aforementioned, there exist numerous variants of scheduling problems (parallel machines, flow-shop, job-shop,... with different constraints such as sequence-dependent setup times, release dates, machine eligibility,... and various objective functions). The best method to solve a scheduling problem depends on the problem's type. Hence to have good solutions, it is necessary to choose a suitable algorithm with good (hyper-)parameters. Hence to customize such production-scheduling part, one must have expert knowledge both on solving methods and on the taxonomy of scheduling problems. For this reason, non-scheduling experts cannot customize such part of the ERP.

In order to solve this problem, some methods have been developed. For example [MPE03] proposed an expert system to solve this. They have a portfolio of 75 algorithms (including dispatching rules, sequencing algorithm, improvement-type algorithms) on the one hand and classification of many scheduling problems on the other hand. They established a mapping between scheduling problems and algorithms based on many scientific publications. Then they develop an expert system, which asks questions to the user (such as: "Does the problem consist of some stages in the series with several machines in parallel at each stage?") which a non-expert can answer, and then, according to the answers, selects the best algorithm to use. The answers help select the type of scheduling problem, and then the mapping selects the algorithm that corresponds to that particular type.

5.1.3 Infologic and Copilote

Infologic is a french society created in 1982 in Valence by André Chabert. Infologic develops and integrates its own ERP system specialized for the agrifood industry. Three ERPs have been developed since the foundation of the company: *Agro V1*, *Agro V2* and *Copilote*. As technologies evolved, *Agro V1* and *Agro V2* became obsolete and have been replaced with *Copilote* (although few clients still use *Agro V2*). As shown in Fig 5.1, the main strength of Copilote is its large functional perimeter. It integrates classical modules such as commercial management, warehouse management, manufacturing execution, or logistic management. However, Infologic differs from its competitors by integrating some rare native modules in ERP systems. Copilote integrates its own financial and sales forecasts modules, a very powerful

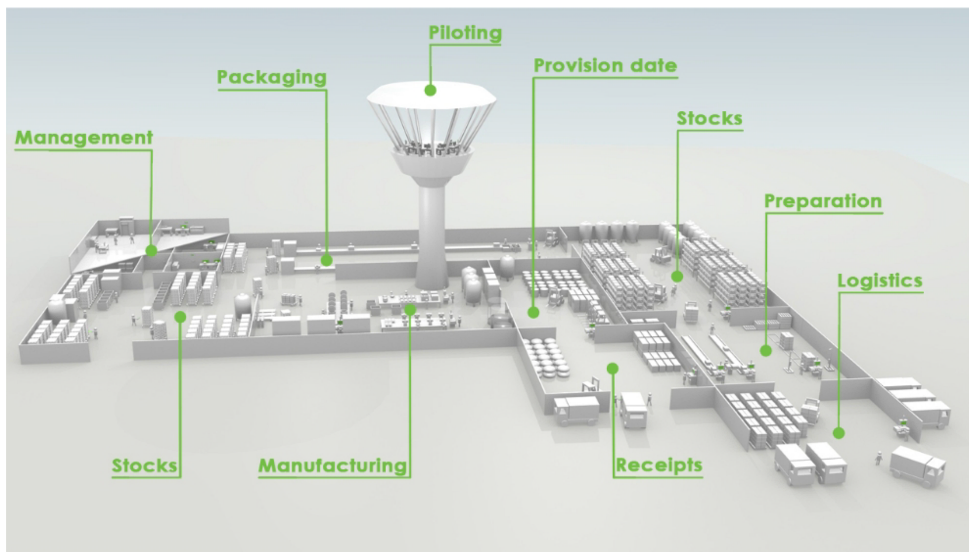


Figure 5.1: Copilote functional perimeter

decision-making system that monitors a company's real-time activities, and an efficient Electronic Data Interchange tool. Furthermore, as aforementioned for ERP, *Copilote* is highly customizable, and it has more than 5500 general parameters that allow configuring modules.

Scheduling problems arise in different parts of the Infologic client's process. The two main ones are production management and order preparation. The first one is concerned with scheduling the production of the different products that the company sold. The second is concerned with packaging the sold products and their grouping to satisfy customers' orders. Although constraints and objectives are different for these two scheduling problems, and although each warehouse has its own particularities (in terms of constraints or objectives), the idea of Infologic is to develop a unique solver adapted to the different situations.

5.2 Features of the data-set

As aforementioned, the ERP *Copilote* is deployed on many client sites. Each site has its own working processes. In particular, different scheduling models are used to represent the different warehouses. The objective is to develop a generic solver which is able to solve all the scheduling problems met by *Copilote*'s users. The modeled problems correspond to order preparation in the food industry. This preparation is done in two steps: in the first step, products are collected in the stock (this is the picking part), and in the second step, they are weighted (in the food industry, products are often sold according to their weight), packaged, and all products of the same order are put on the same pallet (this is the composting part). Different workers generally do these two steps. Often, a first-team collects the products in the stock and brings them close to the team that weight, package and gather them. The second team's work is longer than the first team's work. Hence the operations are only sequenced for the second team, and the schedule of the first team is deduced from the one of the second team (the second team is the bottleneck of the whole process). Depending on the warehouse, some constraints are modeled or not.
Some people want to model the time taken to change a packaging roll or the time taken to adjust the balance (which is used to weight the product). In such a case, we use sequence-dependent setup-times to model it. Hence, the setup times considered here have a particular structure. Each job has two characteristics (namely, the product and the packaging roll used for the job). When two jobs are sequenced one after the other on the same machine, if they use different products, a setup time of 16 seconds must be applied, and if they use different packaging rolls, an additional setup time of one

minute must be applied. Hence the setup-times take values in $\{0, 16, 76\}$.
In other warehouses, managers want to consider that workers have breaks at different time intervals; in that case, we consider scheduled breakdowns on machines. In other cases, it is necessary to take into account the fact that not all workers work at the same speed (because of their competences or because of the tools they use).
Furthermore, in order preparation, each order comprises several products that must be prepared. However, products of the same order are sent together to the client. So they must be put on the same pallet to be shipped together. Hence, once we start preparing an order, we must put a pallet on the ground, and this pallet remains on the ground until all the orders' products are completed. The available physical space on the floor is limited, and thus the number of orders which can be simultaneously underway is limited. The *Group Cumulative* constraint is used to model this situation.

This work evaluates the interest of a using a generic solver. We have collected data on 548 days of works. Each day of work consists of a given number of jobs with their duration, release dates, due dates, and the characteristics needed to calculate the sequence-dependent setup-times. We also have information on the machines: their speed and their calendar. These instances have been adapted, and some constraints are considered or not according to the problem we are evaluating. Hence the same instances are used for the different problems, but some information are ignored in some cases (for example, when we consider a problem in which we try to minimize the makespan, we do not consider the due dates).

In the collected data, the information about the GC were missing. Hence, in order to evaluate the influence of the tightness of the GC constraint, we generate two instances for each instance in our data-set. For each raw instance, an upper bound (denoted x) on the number of active groups is computed as follows: first, a greedy algorithm (ATCS) is used to compute a solution s for the $Q|r_j, brkdwn, s_{jk}|\sum T_j$ problem (without the GC constraint); then x is assigned to the maximum number of active groups during the whole time horizon in s. As our goal is to study the impact of GC on the solution process, we consider two classes of instances: in the first class, denoted *loose*, the limit l is set to $l = 0.7 * x$ and in the second class, denoted *tight*, l is set to $0.3 * x$. We use the notation GC_{loose} (resp. GC_{tight}) to indicate that we consider a variant of the $GCSP$ where the GC constraint is loose (resp. tight).

To evaluate the impact of the GC constraint (and the tightness of the bound l) on the solution process, in Fig. 5.2, we display the evolution of the cumulative number of solved instances with respect to time for the scheduling problem $Q|r_j, s_{jk}|\sum T_g$, where we consider that an instance is solved when CPO has found a solution and proved that this solution is optimal. In this case, CPO is able to solve 360 instances (among the 548 instances of the benchmark) within one hour. We also display results on the same set of instances when adding GC for the two classes (which only differ on the value of l). Clearly, GC increases the hardness of the problem, and increasing the tightness of GC (by decreasing the limit l) also increases hardness: 318 (resp. 285) instances are solved within one hour for the loose (resp. tight) class.

All the considered scheduling problems are variants of the *uniform parallel machine scheduling problem* (problem $Q|\beta|\gamma$ in the Graham notation). Some problems can be considered as identical parallel machine scheduling problems ($P|\beta|\gamma$), but they are just a special case of $Q|\beta|\gamma$ where all the machines have the same speed.). However, constraints (β field) and objective functions (γ field) vary. Hence, by combining constraints and objective functions, we consider the ten following scheduling problems:

1. $P||C_{max}$
2. $P|r_j|\sum T_j$
3. $Q|r_j|\sum T_j$
4. $Q|r_j, s_{jk}|\sum T_j$
5. $Q|r_j, brkdwn|\sum T_j$
6. $Q|r_j, brkdwn, s_{jk}|\sum T_j$
7. $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$
8. $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$

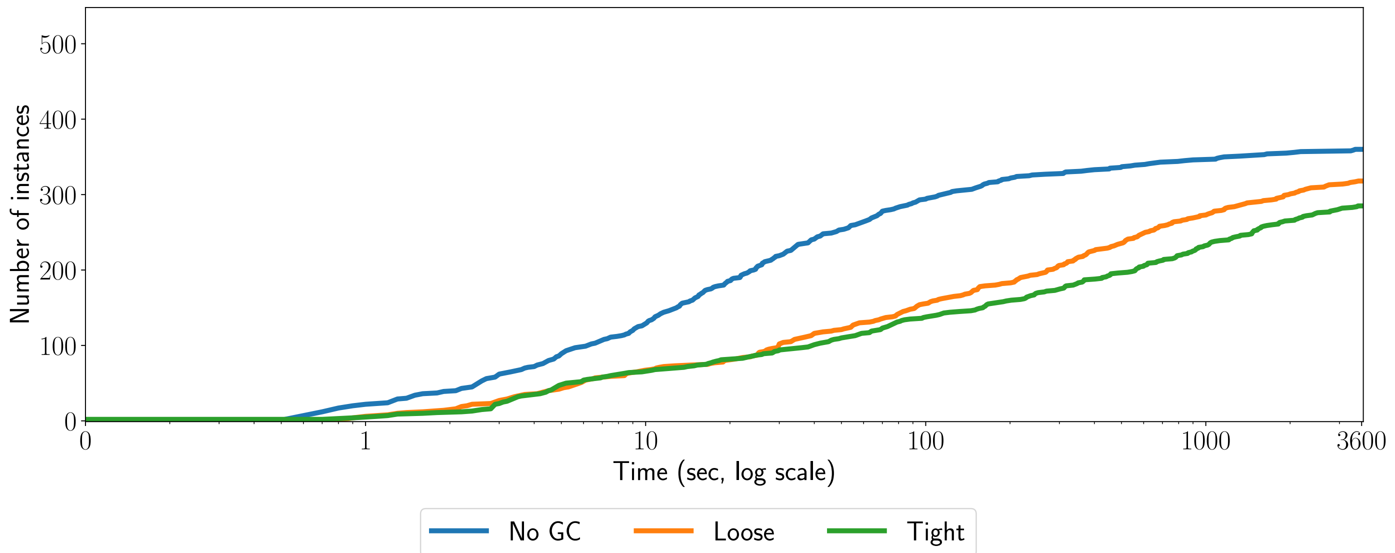


Figure 5.2: Evolution of the cumulative number of instances solved by CPO with respect to time for the GCSP when l is loose or tight.

9. $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$
10. $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

The six first considered problems are described precisely in sec 1.2.5. Problems 7 and 8 correspond to Problem 4 with an additional GC constraint, as defined in Chapter 3 (Definition 3.1.1), where $l = 0.7x$ in Problem 7 and $l = 0.3x$ in Problem 8. Similarly, Problems 9 and 10 correspond to Problem 6 with an additional GC constraint.

Fig 5.3 shows the distribution of instances according to the number of jobs and number of machines. Some additional statistics on the data-set can be found in table 5.1

For the start of the machines, they start either at six or at eight. Among the 2233 machines (in all instances), 2121 start at six, and 112 start at eight.

5.3 Discussion

In this chapter we give a review of industrial applications of scheduling theories. In particular we describe some context where applying scheduling theories leads to substantial benefits for industrial companies. We also introduce our industrial context and especially the Infologic company. We describe how a scheduling module in an ERP is confronted to many different scheduling problems and we present existing approaches to ease customization of ERPs. For a scheduling module in an ERP, being able to adapt the algorithms to the different scheduling problems is crucial. This adaptation will be studied more deeply in chapter 7. In the last section of this chapter, we give some relevant figures with regard to our benchmark. In particular we give some statistics (mean, median, min, max, standard deviation) for different quantities characterizing our instances: number of jobs, number of machines, number of groups, ...
In the next chapter, we will evaluate the methods described in Chapter 2 and 4 on this new benchmark. In particular, by adding or removing some constraints we will be able to evaluate the impact of these constraints on the methods used to solve the problems.

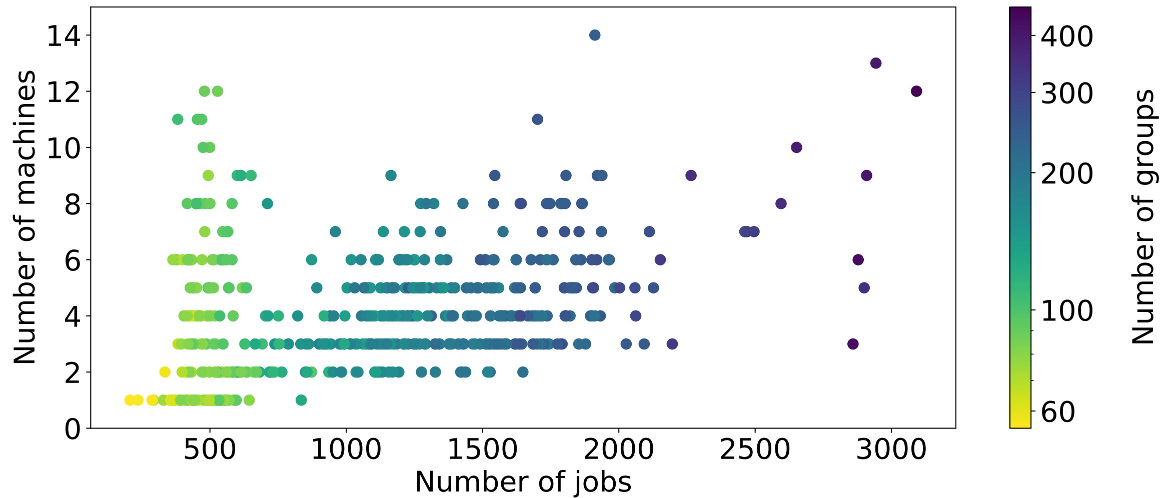


Figure 5.3: Distribution of instances according to the number of jobs and number of machines

	Mean	Median	Standard deviation	Min	Max
Number of jobs	1112.7	1136.5	546.42	207	3092
Number of machines	4.07	4	2.22	1	14
Number of groups	163.4	169	70.2	55	462
Jobs duration (sec)	97.35	36.0	250.5	3	5400
Release date	06:35:21	09:01:00	04:49:17	00:00:00	15:24:00
Due dates	14:38:58	14:00:00	03:02:34	09:00:00	23:45:00
Machines speed	1.08	1.1	0.05	0.7	1.4

Table 5.1: Statistics on the used data-set.

Chapter 6

Experimental Results

Contents

6.1 Experimental settings **91**
6.1.1 Algorithms and parameters' choice 92
6.1.2 Performance measures 95
6.2 Individual problems' results **96**
6.2.1 Assignment problem $P||C_{max}$ 98
6.2.2 Release date and due date : $P|r_j|\sum T_j$ 100
6.2.3 Speeds : $Q|r_j|\sum T_j$ 102
6.2.4 Sequence dependent setup times : $Q|r_j, s_{jk}|\sum T_j$ 104
6.2.5 Breaks : $Q|r_j, brkdwn|\sum T_j$ 106
6.2.6 Breaks and setup times: $Q|r_j, brkdwn, s_{jk}|\sum T_j$ 108
6.2.7 GC_{loose} no breaks: $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$ 110
6.2.8 GC_{tight} no breaks: $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$ 112
6.2.9 GC_{loose} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$ 114
6.2.10 GC_{tight} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$ 116
6.3 Results over all problems **117**
6.4 Discussion **121**

In this chapter we give our experimental results. We first introduce our experimental settings in section 6.1. In particular, we describe the used algorithms and the considered performance measures. By considering different constraints for a same data-set, we can evaluate the impact of each constraint on scheduling problems. In section 6.2 we give the results for each studied scheduling problem. We also give some explanations and comments about these results. In particular we show that some methods can handle some constraints easier than others. Finally in section 6.3 we give an overview of the results for all the considered scheduling problems. This allows the reader to easily compare the results over the different scheduling problems.

6.1 Experimental settings

All experiments reported in this work have been performed on a processor Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with up to 16 GB RAM[1]. When an experiment exceeds 16GB RAM, it is stopped. Depending on the scheduling problem, statistics on the RAM consumption are given to evaluate the different approaches' memory costs.

[1]We gratefully acknowledge support from the CNRS/IN2P3 Computing Center (Lyon - France) for providing computing and data-processing resources needed for this work.

The different studied problems are precisely described in the previous chapter. There are 10 scheduling problems, variants of the uniform parallel machine scheduling problem.

6.1.1 Algorithms and parameters' choice

In this subsection, we describe precisely all the settings used for the different algorithms. The considered algorithms can be gathered into six families: Tabu search, Ant Colony Optimization (ACO), Constraint Programming (CP), Mixed Integer Linear Programming (MIP), ACO-Tabu and CPO-ACO. For each algorithm, we have tested the influence of several parameters, in order to select the best configuration.

Tabu search

Our Tabu algorithm (described in Section 2.3) is implemented using the Java language. Hyper-parameters and parameters have been set as follows:

- We tried four different neighborhoods: *insert*, *swap*, *insert most late job*, and *swap most late job* (as described in section 2.3.2). We observed that *insert* and *swap* often spend a lot of time at evaluating neighbors which do not change the value of the objective function. *insert* and *swap* favor diversification whereas *insert most late job* and *swap most late job* favor intensification. In our experiment, the two latter neighborhoods often lead to better results, and we report results obtained with these two neighborhoods only.
- We tried three different strategies to fill tabu lists: preventing a job from going back to a position it has already occupied, preventing a job from going back to a machine it has already occupied and preventing a job from moving. Theses strategies have performance which are correlated with the used neighborhood. Indeed, some strategies work well with some neighborhoods, whereas they have terrible performance with other neighborhoods. We observed that preventing a job from going back to a position it has already occupied is the best strategy when combined with the *insert* and the *swap* neighborhoods. On the other hand, when using the *insert most late job* and the *swap most late job* neighborhoods, the best results are obtained when we forbid a job from moving again. The latter can be explained as follows: *insert most late job* and *swap most late job* favor intensification, as the set of candidates which can be selected to move is small (only the one which have the greatest tardiness). On the other hand, preventing a job from moving favors diversification. Indeed if the jobs with the greatest tardiness are tabu, then it allows to select jobs with lower tardiness. Hence combining these neighborhoods with this strategy is a good compromise between intensification and diversification.
- We evaluated different strategies for managing the tabu list length. First we tried to fix the tabu list size, and we explored different values in the range [3, 30]. We observed that the best value depends on the instance. Then we tried another approach which consists in having a tabu list size which is function of the number of jobs in the instance. Hence the size of the tabu list is equal to $x * |\mathcal{J}|$, and we tried values for x in the range [0.2%, 2%] (for an instance with 1000 jobs, it corresponds to a size in [2, 20]). Finally we tried to have a dynamic tabu list' size. We have an additional parameter $coeffNoImprov$. Each time we realized $coeffNoImprov * size$ iterations, where $size$ is the tabu list' size, without improving the best solution, we increase the tabu list' size of $x * |\mathcal{J}|$. Once we find a new improving solution, we reset the size to its initial value. The best results were obtained with this dynamic strategy.
- When evaluating the neighbors of a solution, we can choose to stop the iteration search as soon as we find an increasing neighbor. On the other hand, we can also choose to always evaluate the entire neighborhood and select the best neighbor of the neighborhood. Once again, this choice is highly correlated with the neighborhood choice. Indeed when considering large neighborhoods (such that *insert* and the *swap* ones), selecting the first improving neighbor leads to better results. On the other hand, with smaller neighborhoods, evaluating the whole neighborhood and selecting the best neighbor often leads to better results.

We finally consider two *tabu search* variants with the following parameters: we forbid to move a job which has already been moved. The tabu list size is set to $0.005|\mathcal{J}|$, and we take $coeffNoImprov = 5$, i.e., each time we have done

$5 * tabuListSize$ iterations without improvement, we increase the size of the tabu list. Finally, we always select the best neighbor of the neighborhood. We consider two variants with two different neighborhoods:

- $Tabu_I$ with *insert most late job* neighborhood (in $P||C_{max}$ (which do not consider tardiness) we consider the *insert* neighborhood);
- $Tabu_S$ with *swap most late job* neighborhood (in problem $P||C_{max}$, we consider the *swap* neighborhood).

ACO

Similarly to Tabu search, our ACO algorithm is implemented in Java. It has a lot of parameters to adjust:

- α and β parameters. For these two values we tried different combinations in the range [1,10]. We observed that higher values for β often lead to better results. Indeed, given two values $\eta(j)$ and $\eta(j')$ for two jobs j and j' with $\eta(j) > \eta(j')$, the bigger β, the more we increase the probability of selecting j rather than j'. The best value for α is highly correlated with other parameters. A higher value for α will favor intensification (indeed when α is greater we give more weight to trails which have already been rewarded). Hence when we combine ACO and another method (*tabu search* or *CPO*), it is valuable to have a high value for α. Indeed in such a case, we spend less time in rewarding solutions (as we spend time in the other method). So, it is interesting to increase intensification. On the other hand, when ACO is not combined with another method, lower values for α often lead to better results, as the diversification phase is more important.
- We considered both $MMAS$ and $P-ACO$. In our preliminary experiments, we observe that both systems have comparable results, with a small advantage for $MMAS$. Hence we decided to focus on $MMAS$
- The values for ρ, τ_{min} and τ_{max}. ρ has a role similar to the one of α, when increasing ρ we favor intensification, whereas when we decrease it we favor intensification. We try for ρ several values in the range [0.02, 0.3]. Once again, the best value for ρ is not the same if we combine ACO with another approach or not. For τ_{min} and τ_{max} we tried the two different strategies described in section 2.2 (manually setting them or setting τ_{max} to $\frac{1}{f(x_{best})}$, τ_{min} to $\frac{\tau_{max}}{5}$). When manually setting them, we tried values in the range [0.0001, 4]. We found that manually setting them often lead to better results. It is important to notice that, in such a case, in order to have a pheromone factor on a same scale than the heuristic factor, we normalize both of them. It means that when we compute $f_\tau(j)$ and $\eta(j)$, we set $f_\tau(j) \leftarrow \frac{f_\tau(j) - \min_{j'} f_\tau(j')}{\max_{j'} f_\tau(j') - \min_{j'} f_\tau(j')}$ and $\eta(j) \leftarrow \frac{\eta(j) - \min_{j'} \eta(j')}{\max_{j'} \eta(j') - \min_{j'} \eta(j')}$.
- The pheromone structure (to be chosen among *Position*, *Jobs* and *Time* as presented in section 2.2 and 4.2.3). Depending on the problem, the structures have different performance and none outperforms the others on all problems.

We finally consider the three following variants of ACO:

- ACO_0, which is a *Greedy Randomized Search (GRS). GRS* is a special case of ACO in which we do not consider pheromone trails (*i.e.*, $\alpha = 0$). In this case, the pheromone updating step (lines 5-8 of Algo. 2) is not performed. We take $\beta = 10$;
- ACO_J with *Jobs trails*;
- ACO_P with *Position trails*;
- ACO_T with *Time trails*;

For ACO_J, ACO_P and ACO_T we use the following parameters: $\alpha = 5$, $\beta = 10$, $n_{ants} = 40$, *MMAS strategy*, $\rho = 0.95$, $\tau_{min} = 0.1$, $\tau_{max} = 4.0$

Constraint Programming

We use two solvers for CP: *CP Optimizer (CPO)* and *ORTOOLS*. *CPO* is used with its 12.9.0 version and *ORTOOLS* with its 7.4 version. Both solvers are highly customizable [Man87a]. For *CPO*, we tried to modify the following parameters:

- The search type, where we can choose between *depth-first*, *restart* and *multi-point* (the default being *restart*). We observed that the best results are achieved using the *restart* strategy.
- The inference level of constraints. The possible values are *low*, *basic*, *medium* and *extended*. Depending on the value of the inference level, different propagators are used to tighten variables' domains. We tried the four different values for the constraints *noOverlap* and for the constraint *cumulative* (used in the decomposition of the GC). For both constraints, the best results are obtained with the default value (which is the *basic* inference).

For ORTOOLS we did not try to modify its parameters and we used it with its default values.
Hence we consider the following variants in our experiments:

- CPO which uses the *CP Optimizer* solver and scheduling variables (and can be applied to all problems);
- ORT which uses the *ORTOOLS* solver and scheduling variables (and can be applied to all problems);
- CPO_A which uses the *CP Optimizer* solver and consider a model similar to the one used for *MIP* as described in section 2.5.5 and which is only applied on the assignment problem $P||C_{max}$;
- ORT_A which uses the *ORTOOLS* solver and consider a model similar to the one used for *MIP* as described in section 2.5.5 and which is only applied on the assignment problem $P||C_{max}$;

MIP

We use the CPLEX solver for MIP, with its 12.9.0 version. CPLEX is also highly customizable. However we only consider it with its default values. We consider three variants, with three different models (described in 2.4.2):

- MIP_A uses the model described in Fig. 2.4 and is only applied on the assignment problem $P||C_{max}$
- MIP_2 uses the model described in Fig 2.5 and is only applied on problem $P|r_j|\sum T_j$
- MIP_3 uses the model described in Fig. 2.6 and can be applied to problems $P|r_j|\sum T_j$, $Q|r_j|\sum T_j$ and $Q|r_j, s_{jk}|\sum T_j$

Combining ACO and another approach (CPO-ACO and ACO-Tabu)

As described in section 2.2, ACO is often combined with local search. Furthermore, in section 4.3, we present a new algorithm combining CPO and ACO. In both cases, the combined approach (LS or CPO) is triggered every k ACO cycles. We tried different values for k in the range [1, 20]. The value 5 often lead to good results. Furthermore, in *CPO-ACO*, CPO is executed until it backtracks b times. When combining ACO and *tabu search*, we stop the tabu search when it has done b iterations without improving the incumbent solution. For b we tried different values in the range [100, 10000]. Furthermore, in order to let more and more time to the approach used in combination with ACO, we increase the value of b each time it is restarted. We tried two methods in order to increase it: a geometric increase (with coefficient g, *i.e.*, after each run $b \leftarrow b * g$) and an increase following a Luby sequence [LSZ93]. For *CPO-ACO* we observed that the best results are obtained with the geometric increase with values of g in [10, 20]. When combining ACO and tabu search, we observed that the best results are obtained when not increasing the value of b.

We keep the following approaches for ACO-Tabu:

- $ACO - Tabu_J$ with *job pheromone trails*;
- $ACO - Tabu_T$ with *time pheromone trails*;

where a tabu search is applied every 5 cycles of ACO on the best solution found during the last 5 cycles. The tabu search is stopped when it does 200 iterations without improving its incumbent solution (the value is never increased). For ACO we use the same parameters as ACO_J except that $\rho = 0.8$. Tabu is used with the same parameters as $Tabu_I$

For *CPO-ACO* we consider the following variant denoted $CPO-ACO$: *Jobs trails*, $\alpha = 10$, $\beta = 10$, $\rho = 0.8$, *MMAS strategy*, $\tau_{min} = 0.1$, $\tau_{max} = 4.0$, *CPO* called every 5 cycles, with $b = 1000$ and a geometric progression with $g = 20$;

6.1.2 Performance measures

We denote $\mathcal{I}$ the set of all instances and $\mathcal{A}$ the set of all algorithms. Let $x \in \mathcal{I}$ be an instance, $a \in \mathcal{A}$ an algorithm, and t a time limit. We denote x_a^t the value of the best solution found by a for x within t seconds, and $x^* = \underset{a \in \mathcal{A}}{min}\ x_a^{3600}$ the reference solution, i.e., the value of the best solution found by any of the considered algorithm within one hour.

For some instances, $x^* = 0$, i.e., there exists a solution without tardiness. In this case, we cannot evaluate the quality of x_a^t by computing its ratio to the reference solution (i.e., $\frac{x_a^t}{x^*}$) or its average gap in percentage to the reference solution (i.e., $\frac{x_a^t - x^*}{x^*}$). Hence, we consider the inverse ratio defined as follows:

$$ir_{a,x}^t = \begin{cases} 1 & \text{if } x_a^t = x^* \\ x^*/x_a^t & \text{otherwise} \end{cases}$$

The inverse ratio always belongs to $[0, 1]$ and the closer $ir_{a,x}^t$ to 1, the better x_a^t.

For each run of an algorithm a on an instance x, we measure:

- $ir_{a,x}^t$, for t in $[0, 3600]$;
- the time needed to find the reference solution, denoted $t_{a,x}^{ref}$, i.e., $t_{a,x}^{ref} = \infty$ if $ir_{a,x}^{3600} < 1$; otherwise $t_{a,x}^{ref} = \underset{t \in [0,3600]}{\text{argmin}}\ ir_{a,x}^t = 1$;
- the time needed to find the optimal solution and prove optimality, denoted $t_{a,x}^{opt}$ ($t_{a,x}^{opt} = \infty$ if optimality has not been proven);
- the maximum amount of memory used by the process (the *resident set size*, which corresponds to the portion of memory occupied by the process that is held in main memory (RAM)), denoted $mem_{a,x}$.

For each class of problem $\mathcal{I}$ ($\mathcal{I}$ is a set of instances of a given problem) and each algorithm a, we display the following figures:

- the evolution of the average inverse ratio with respect to time, that plots the evolution of $\frac{1}{|\mathcal{I}|} \sum_{x \in \mathcal{I}} ir_{a,x}^t$ when t ranges from 0 to 3600 (called *Evolution of average inverse ratio* in the figures);
- the evolution of the number of reference solutions found with respect to time, that plots the evolution of $|\{x \in \mathcal{I} | t_{a,x}^{ref} \leq t\}|$ when t ranges from 0 to 3600 (called *Time to reach reference solution* in the figures);
- the evolution of the number of optimality proofs with respect to time, that plots the evolution of $|\{x \in \mathcal{I} | t_{a,x}^{opt} \leq t\}|$ when t ranges from 0 to 3600 (called *Time to prove solutions optimality* in the figures);
- the distribution of $ir_{a,x}^{60}$ for all x in $\mathcal{I}$ (called *Inverse ratios distribution after one minute* in the figures);
- the distribution of $ir_{a,x}^{3600}$ for all x in $\mathcal{I}$ (called *Inverse ratios distribution after one hour* in the figures);
- the distribution of $mem_{a,x}$ for all x in $\mathcal{I}$ (called *Distribution of the maximum RAM value* in the figures).

In boxplots, the orange line corresponds to the median, the rectangle corresponds to the first and the third quartile, and the whiskers correspond to the first and the ninth decile. The dots are outliers.

It is important to notice that the considered approaches can be split into two categories. On the first hand there are the exact approaches (*MIP*, *CP*, *CPO-ACO*) which can give optimality proofs. On the other hand there are the heuristic approaches (*Tabu*, *ACO*, *ACO-Tabu*) which cannot give optimality proofs, except in the trivial case where they found a solution with zero tardiness. In order to allow heuristic approaches to prove optimality for some solutions, at the beginning of the search we compute a lower bound on the objective function (see section 1.2.5). This bound allows us to prove solution optimality. Indeed, as soon as a solution whose value is equal to the bound is found, we know that this solution is optimal. Hence the $t_{a,x}^{opt}$ measure is considered for all approaches (exact and heuristic).

In Section 6.2, we will discuss the results problem after problem. In section 6.3 we will display only a part of the results but for all problems. It allows the reader to compare the evolution of the results over the different problems.

6.2 Individual problems' results

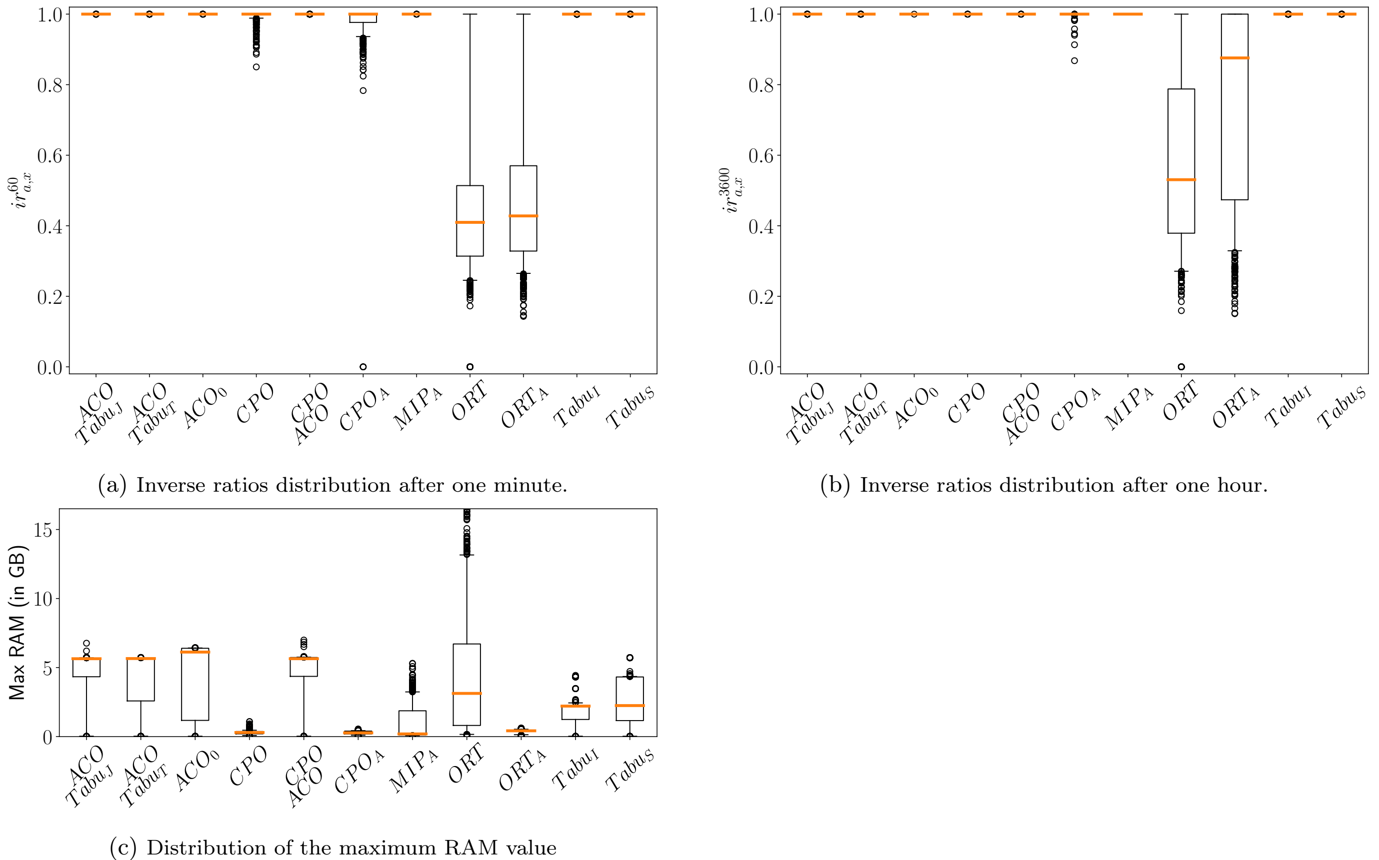
In Section 6.2.1 to 6.2.10, we report and analyze results obtained by the 12 algorithms on our 10 different scheduling problems. For each problem, we first list the algorithms that are considered in the study (because in some cases some variants are either meaningless, or not competitive at all). Then, we display the inverse ratio distributions after one minute and one hour of CPU time and the memory consumption distribution for all the considered algorithms. This allows us to discard some algorithms, either because they are not competitive, or because they have performance similar to other algorithms. Finally, we display more detailed results for the algorithms that are not discarded, *i.e.*, the evolution of the inverse ratio, the number of reference solution found and the number of optimality proofs with respect to time.

(Page intentionally left blank in order to have figures and results analysis side by side)

6.2.1 Assignment problem $P||C_{max}$

Considered algorithms: As mentioned in Section 1.2.5, for problem $P||C_{max}$ we are only interested in knowing which job is assigned to which machine, and we do not care about the order of jobs on the different machines, or the time at which they are scheduled. Hence pheromone structure *Jobs*, *Position* or *Time* cannot help improving the search. Hence we only consider the ACO_0 variants in this section. For $ACO - Tabu$ and $CPO - ACO$ pheromone is not used neither. For this problem the used dispatching-rule is the *longest processing time first* procedure.

Global analysis: Fig 6.1 shows the results for all the considered algorithms. First of all, we notice that ORTOOLS has the worst result on that problem. Its mean inverse ratio is equal to 0.53 after one hour with scheduling model and to 0.88 with assignment model. It highlights the importance of the model. We also notice that the median of the maximum RAM consumed for each instance is ten times greater for the scheduling model than for the assignment model.
For *CPO*, we notice that the results are slightly better with the scheduling model than with the assignment model.
$ACO - Tabu_J$ and $ACO - Tabu_T$ have similar performance, and they are both slightly outperformed by $CPO - ACO$. Hence, we do not report detailed results obtained by $ACO - Tabu_J$ and $ACO - Tabu_T$. Also, we do not report detailed results for $Tabu_J$ (because it is very similar to $Tabu_S$), ORT and ORT_A (because they are strongly outperformed by other approaches), and CPO_A (because it is outperformed by CPO).



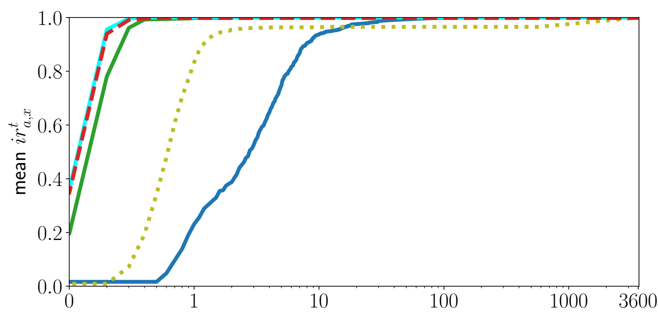
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

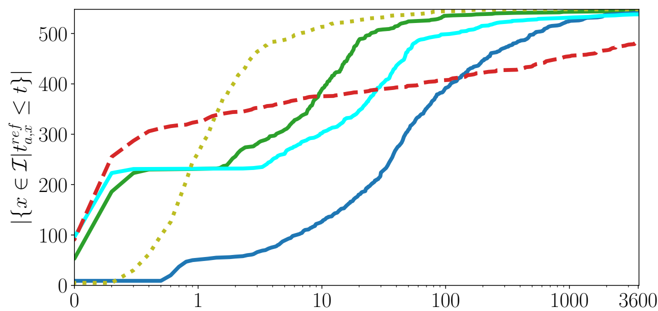
(c) Distribution of the maximum RAM value

Figure 6.1: Results of all algorithms for $P||C_{max}$

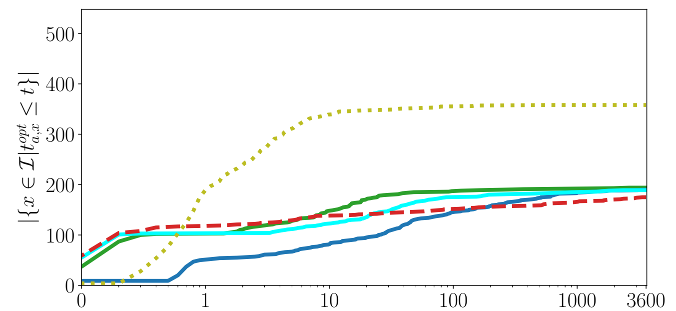
Detailed results analysis: Fig 6.2 shows the detailed results for the selected algorithms. We notice that MIP_A outperforms other methods on that problem for time limits greater than one second: it finds the best solution on every instance, and it finds it on 539 (98.4%) instances in less than one minute. It also proves optimality for 358 instances.
ACO_0 is also well suited for this problem as it finds the best solution on 547 (99.8%) instances within one hour and on 525 (95.8%) instances within one minute. However, ACO_0 proves the optimality of solutions on fewer instances than MIP_A (194 instances, 35.4%).
CPO also finds the reference solution on almost every instance (541, 98.7%) within one hour. However, there are many instances for which it needs much more time to find this reference solution than MIP_A. For example, to find the reference solution on 539 instances, it needs 2597 seconds (whereas it needs one minute to MIP_A to do so).
Finally, $Tabu_S$ has good results for really short time limits. It finds the reference solution on half of the instance in less than 0.3 seconds and it has a mean inverse ratio greater than 0.99 in the same time. However, for longer time limits, the results are less impressive. After one hour it finds the reference solution on 483 (88%) instances.
$CPO - ACO$ have performance slightly worse than those of ACO alone.



(a) Evolution of average inverse ratio



(b) Time to reach reference solution



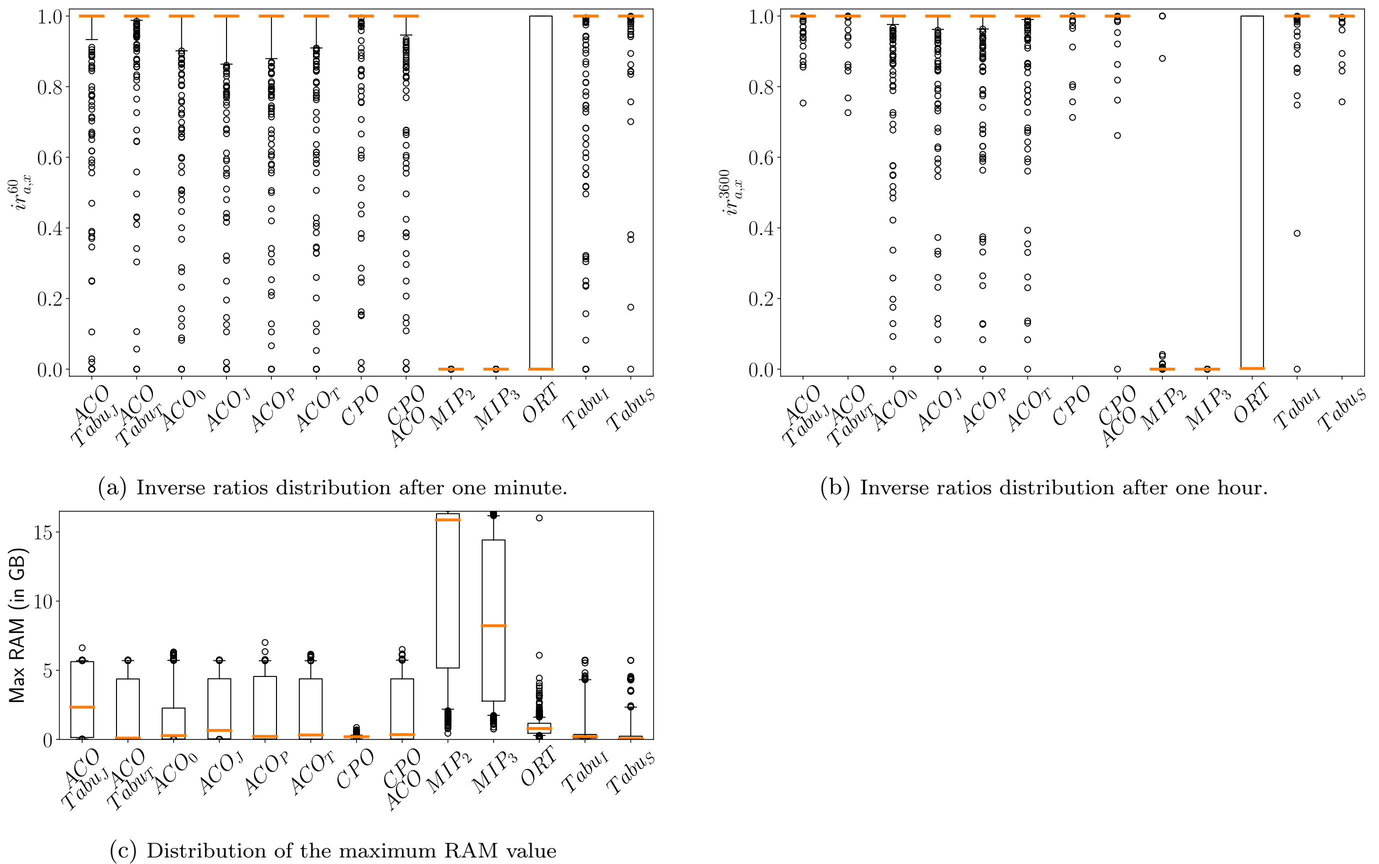
(c) Time to prove solutions optimality

Figure 6.2: Detailed results for $P||C_{max}$

6.2.2 Release date and due date : $P|r_j|\sum T_j$

Considered algorithms: All the algorithms are considered for this problem (except the variants of *MIP* and *CP* whose models do not apply). For this problem which does not consider sequence-dependent setup-times, we use the *ATC* rule in order to greedily compute solutions.

Global analysis: Fig 6.3 shows the results for all the considered algorithms. The first noticeable result is that both Linear Programming models have terrible performance on that problem (using the CPLEX solver). With the second model, the reference solution is found on only three instances (and the proof of optimality is given), and for the first model, the reference solution is not found on any instance. Furthermore, CPLEX needs far more memory than the other methods (with the second model needing even more memory than the first one). It is important to notice that [FL13] (from which the second model was adapted) solve instances which never have more than 50 jobs and five machines. In our case, the number of jobs is often greater, which can explain the difference in terms of results. Hence we do not report detailed results for MIP models.
All *ACO* variants have comparable performance (with ACO_T slightly better as its first decile of the inverse ratio after one hour is equal to 0.99 (against 0.97 for ACO_0, 0.96 for ACO_J and 0.96 for ACO_P)). Hence we only keep ACO_T for the detailed analysis. Similarly both *Tabu* methods have similar performance. It is noticeable that they both find the reference solution on more than 90% of instances in less than one minute. Hence, we report detailed results only for $Tabu_I$. *ORT* is always outperformed by *CPO*. Hence, we do not report detailed results for *ORT*. Once again $ACO-Tabu_J$ and $ACO-Tabu_T$ have similar performance, and they are both slightly outperformed by $CPO-ACO$.



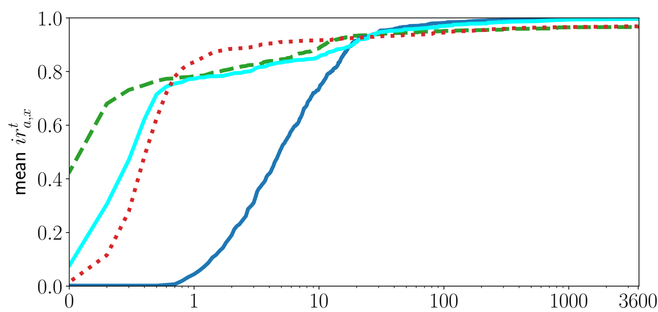
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

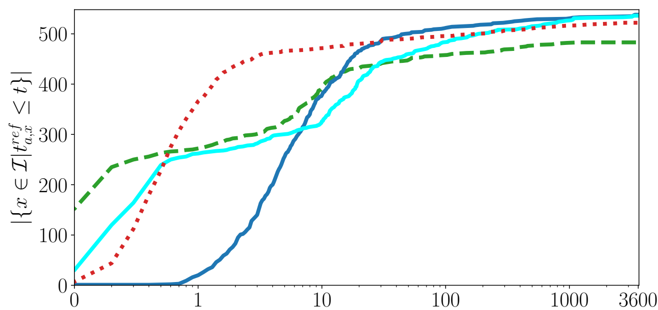
(c) Distribution of the maximum RAM value

Figure 6.3: Results of all algorithms for $P|r_j|\sum T_j$

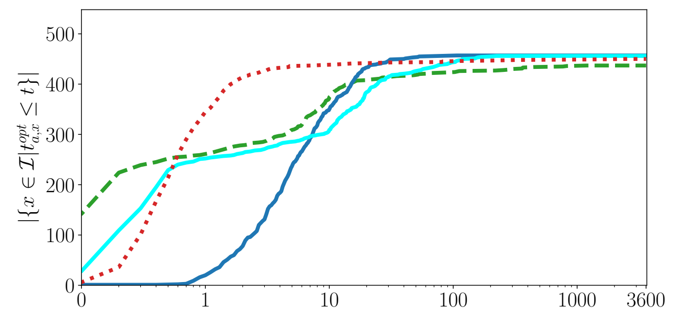
Detailed results analysis: Fig 6.4 shows the detailed results for the selected algorithms. Constraint Programming achieves the best results on that problem in terms of number of instances on which optimality is proved when using the CP Optimizer solver (optimality proved on 457 instances, 83.4%).
$Tabu_I$ is especially well-suited for this problem as it proves optimality on 450 instances (82.1%). The local search's real strength for this problem is the time at which it finds good solutions. It takes 1.5 seconds for the $Tabu_I$ algorithm to find the reference solution on 75% of instances (the second best method that finds the reference solution on 75% of instances is CPO in 13 seconds).
Finally, ACO_T also has good results, but it finds the reference solution on fewer instances (483 (88.1%) against 538 (98.2%) for CPO, for example).
For time limits lower than 30 seconds, $CPO - ACO$ has performance comparable to those of ACO (or slightly worse). However, after 30 seconds, it becomes better than ACO and after one hour it has performance comparable to those of CPO (reference solution found on 536 instances (97.8%) and optimality proved on 456 instances (83.2%)). It offers a good compromise as it is competitive with heuristic approaches for short time limits, and it is competitive with CPO for long time limits.



(a) Evolution of average inverse ratio



(b) Time to reach reference solution



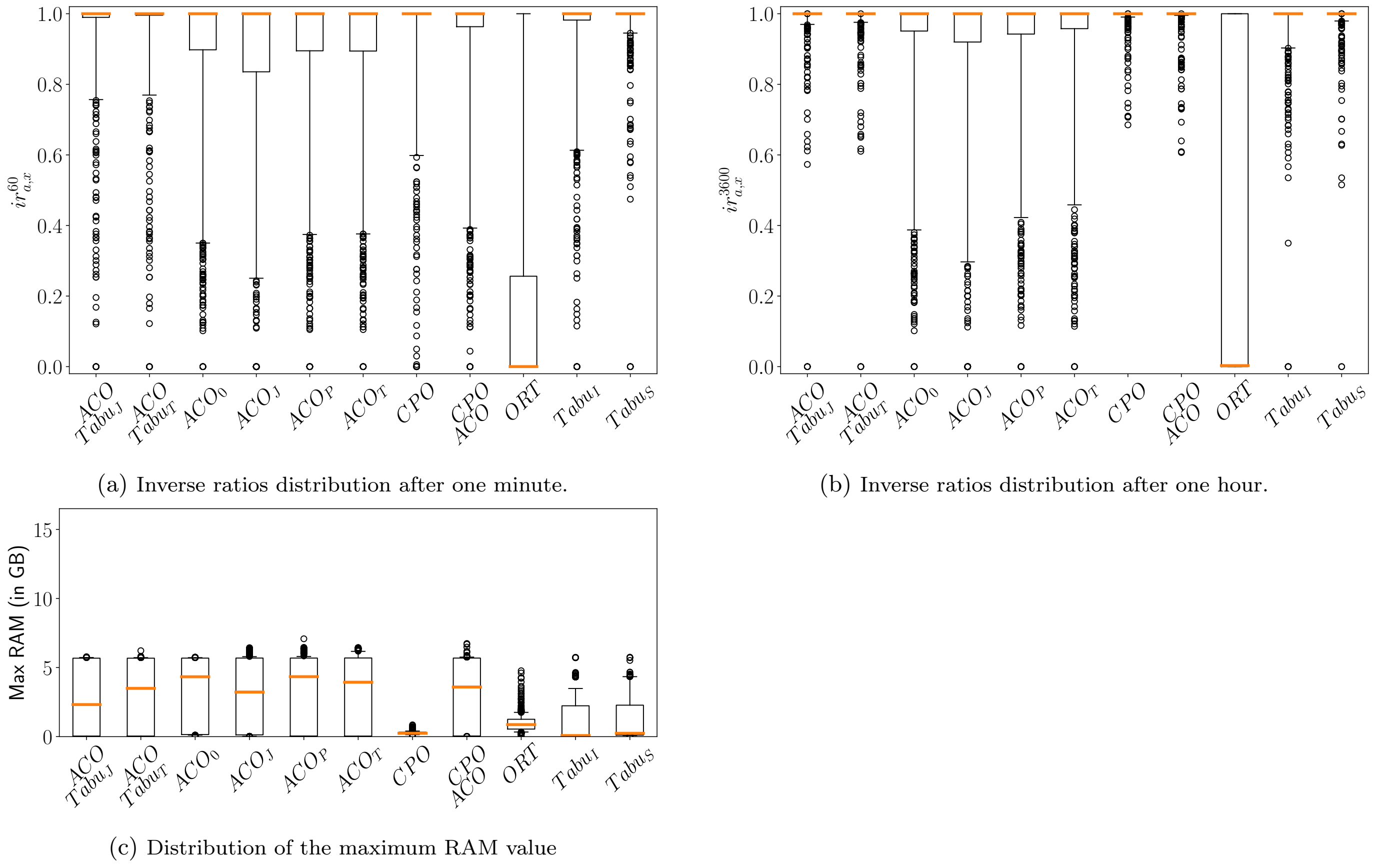
(c) Time to prove solutions optimality

Figure 6.4: Detailed results for $P|r_j|\sum T_j$

6.2.3 Speeds : $Q|r_j|\sum T_j$

Considered algorithms: As *Linear Programming* has difficulties solving the previous problem, we do not consider it anymore.

Global analysis: Fig 6.5 shows the results for all the considered algorithms. The results are similar to those of the previous section, and hence we consider the same algorithms for the detailed analysis. We can notice that *ACO* variants have results worse than those of the previous section.



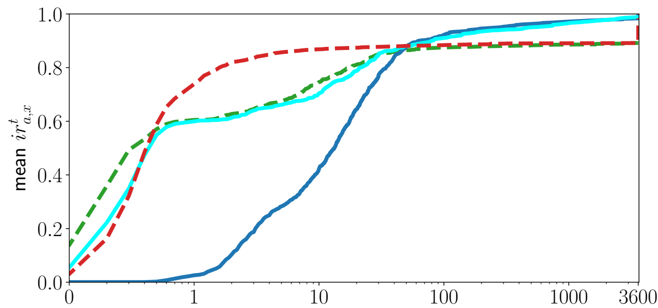
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

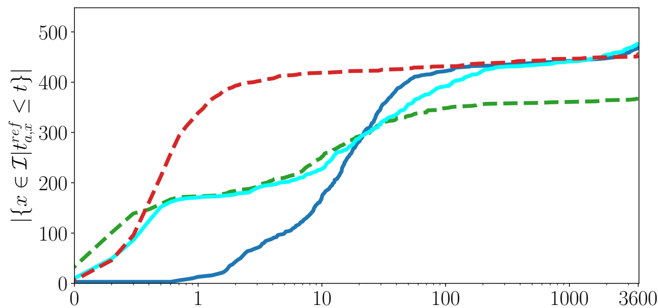
(c) Distribution of the maximum RAM value

Figure 6.5: Results of all algorithms for $Q|r_j|\sum T_j$

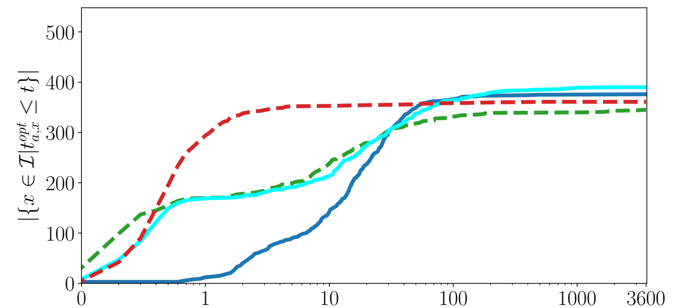
Detailed results analysis: Fig 6.6 shows the detailed results for the selected algorithms. For methods considered in this section and in the previous one (without speeds on the machine), the most noticeable difference is that the gap between methods is more significant on that problem. The gap in terms of percentage of instances on which the reference solution is found between CPO and $Tabu$ was 0.2% in the previous section, whereas the gap is 1.8 % in this case. All the same, the gap between $Tabu$ and ACO was 9.5% in the previous problem and is now 14.2%.
Combining ACO and CPO allows to takes advantage of both worlds. At the start of the search (in the ten first seconds), it quickly finds good solutions (as ACO_T), and for longer searches, it finds solutions as good as those found by CPO. Furthermore, this hybrid method is the one which is able to give the higher number of optimality proofs (390, 71.2%, against 376 for CPO, 68.6%).



(a) Evolution of average inverse ratio



(b) Time to reach reference solution

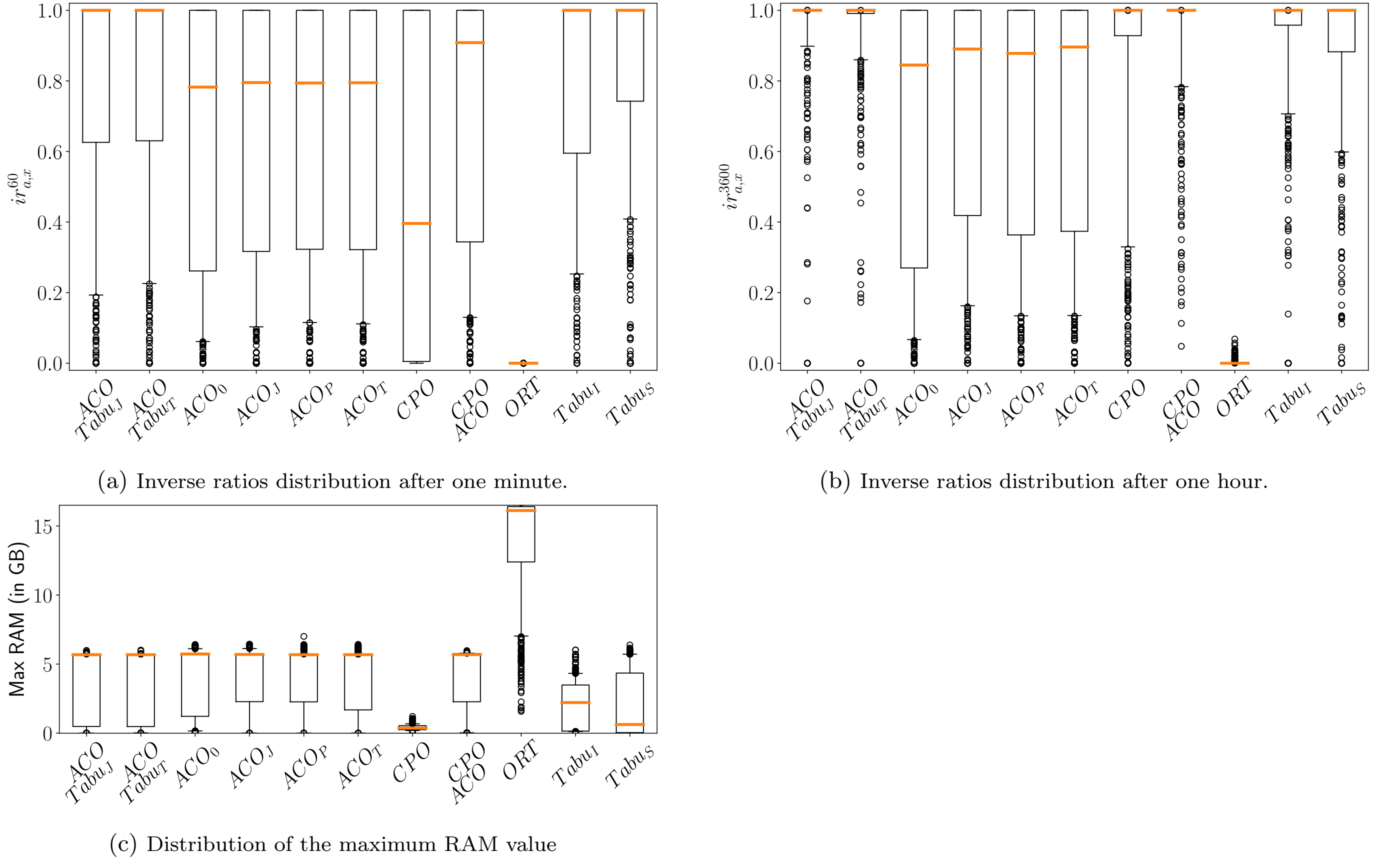


(c) Time to prove solutions optimality

Figure 6.6: Detailed results for $Q|r_j|\sum T_j$

6.2.4 Sequence dependent setup times : $Q|r_j, s_{jk}|\sum T_j$

Considered algorithms: The algorithm considered in this section are the same as in the previous section for the same reasons. For this problem which do consider sequence-dependent setup-times, we use the $ATCS$ rule in order to greedily compute solutions.



(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

(c) Distribution of the maximum RAM value

Figure 6.7: Results of all algorithms for $Q|r_j, s_{jk}|\sum T_j$

Global analysis: Fig 6.7 shows the results for all the considered algorithms. The first noticeable result for this section is that ORT cannot find the reference solution in any instances. Sequence-dependent setup times consume much memory when using ORT, and it exceeds the max memory limit (16GB) on more than half of instances. The best-found solution on any instance has an inverse ratio equal to 0.07.

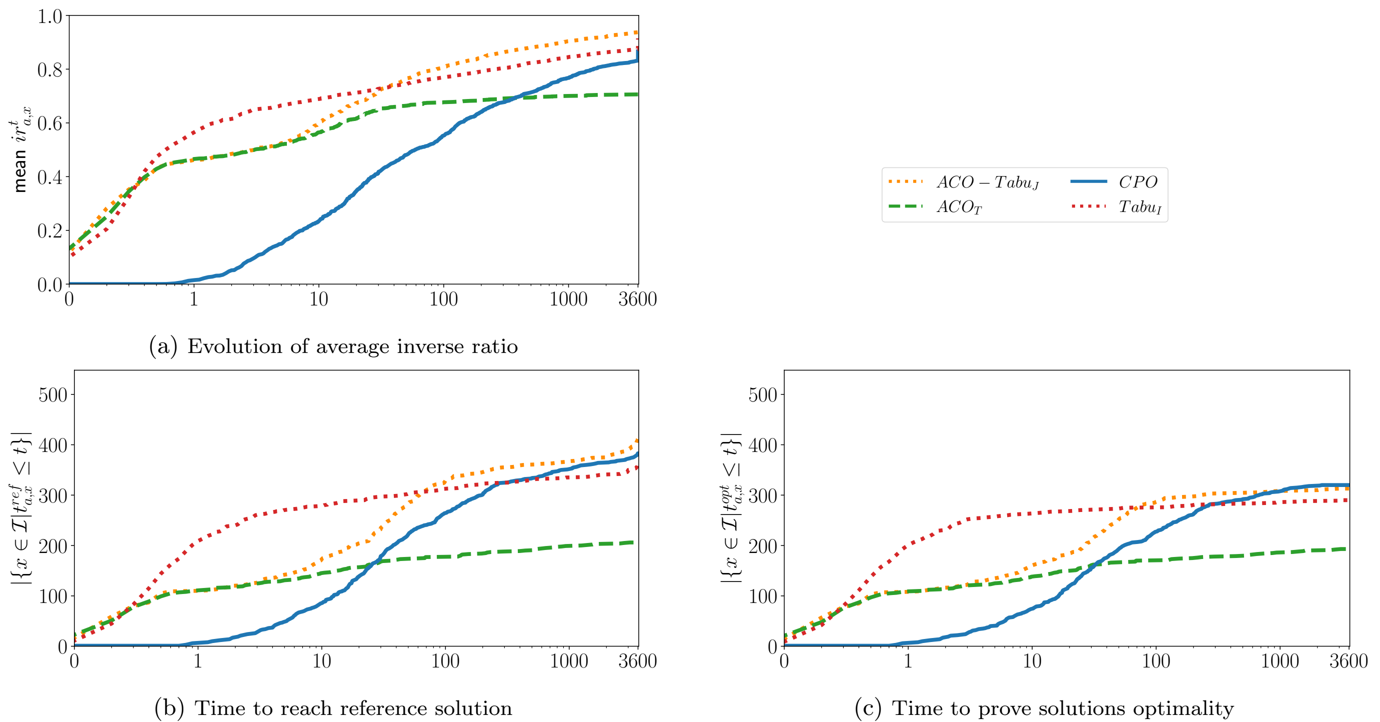
A second noticeable results is that discrepancies exist between ACO variants (this was not the case on the previous problems). In particular the three variants which use pheromone outperform ACO_0. Moreover, we observe that ACO_T has the best median inverse ratio after one hour (0.90 against 0.88 for ACO_P, 0.89 for ACO_J and 0.84 for ACO_0) and ACO_J has the best first quartile inerse ratio after on hour (0.42 against 0.36 for ACO_P, 0.37 for ACO_T and 0.27 for ACO_0). We only keep ACO_T for the detailed results.

We also observe that $Tabu_I$ has better results than $Tabu_S$ (the first quartile inverse ratio after one hour equal to 1 and 0.91 respectively). Hence we keep $Tabu_I$ for the detailed results.

Once again $ACO - Tabu_J$, $ACO - Tabu_T$ and $CPO - ACO$ have similar performance. $CPO - ACO$ finds the reference solution on 416 instances (75.9%) against 414 (75.5%), but $ACO - Tabu_J$ has a better mean inverse ratio after one hour

(0.95 against 0.94), and outperforms $CPO - ACO$ for time limits lower than 2500 seconds. Hence we only displayed the detailed results of $ACO - Tabu_J$.

Detailed results analysis: Fig 6.8 shows the detailed results for the selected algorithms. $Tabu_I$ performs better than CPO on that problem regarding the average inverse ratio after one hour (0.92 against 0.87). However, CPO is able to prove the optimality of solutions on more instances (290 (52.9%) against 320 (58.4%)).
Furthermore, the gap between CPO and $Tabu_I$ on the first hand and ACO_T, on the other hand, is even more significant than in the previous case. Indeed the mean inverse ratio is equal to 0.66 for ACO_T.
The hybrid method $ACO - Tabu_J$ behaves like ACO_T for short time limits (and, hence is worse than $Tabu_I$). However after ten seconds, it becomes better than ACO_T. For time limits greater than one minute it also outperforms $Tabu_I$ for all the considered performance measures. Furthermore, after one hour it is the method that has the best mean inverse ratio (0.95) and which finds the reference solution on the greatest number of instances (414, 75.5%).



(a) Evolution of average inverse ratio

(b) Time to reach reference solution

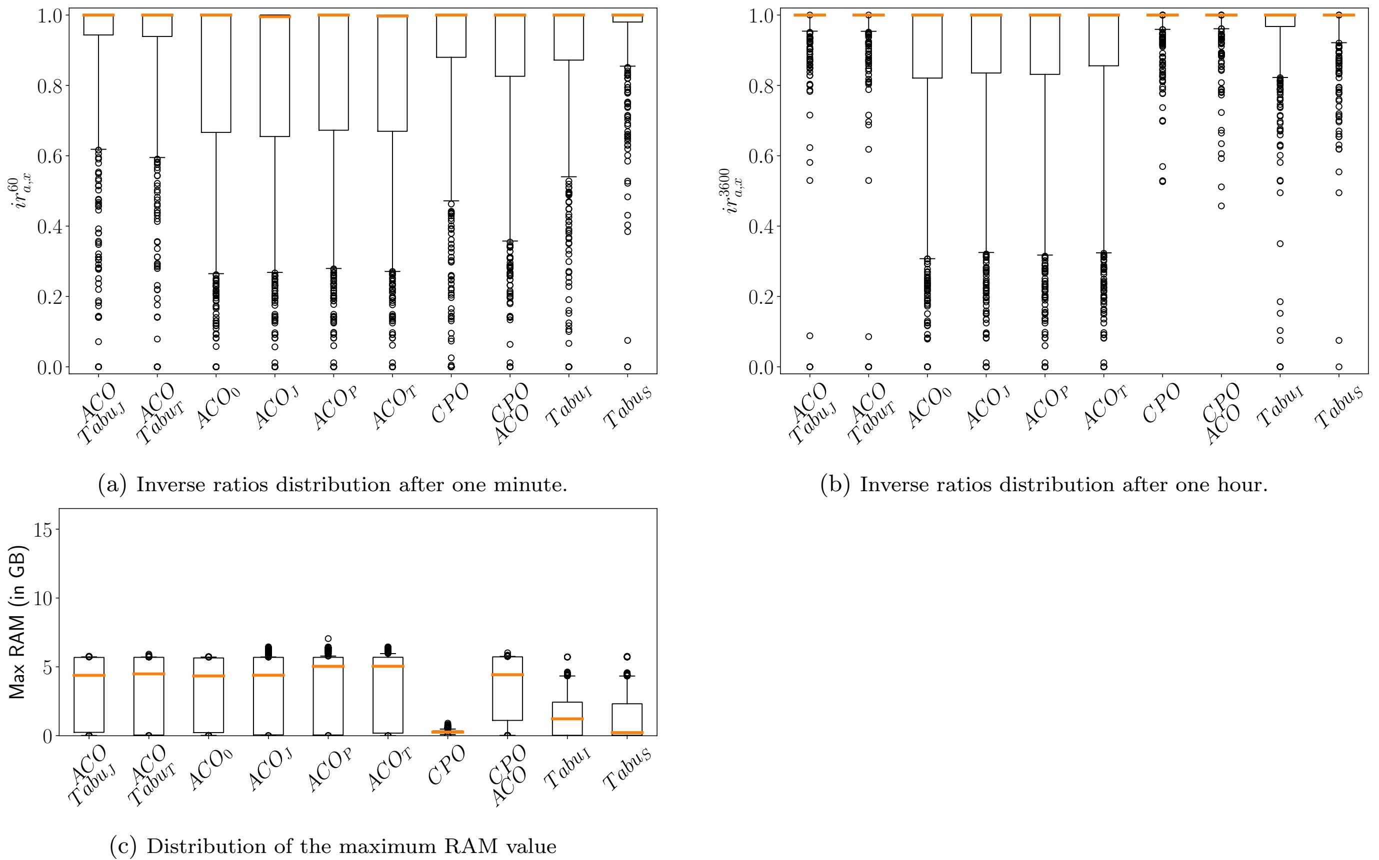
(c) Time to prove solutions optimality

Figure 6.8: Detailed results for $Q|r_j, s_{jk}|\sum T_j$

6.2.5 Breaks : $Q|r_j, brkdwn|\sum T_j$

Considered algorithms: As ORT has generally results worse than those of CPO (and especially on the previous problem), we do not show its results on this problem and the following ones.

Global analysis: Fig 6.9 shows the results for all the considered algorithms. We can notice that there are fewer differences between the considered algorithms than in the previous case. Indeed after one hour, all the algorithms have a median inverse ratio equal to 1, and, except ACO_J and ACO_T, they all have a median inverse ratio equal to one after one minute. We also observe that, contrary to the previous cases, $ACO - Tabu_J$ is the method which has the best first decile (0.98) for the distribution of inverse ratio after one hour. As $Tabu_S$ does better than $Tabu_I$ we only keep it for the detailed analysis. All ACO variants are outperformed by $ACO - Tabu_J$ and, hence their results are not displayed in the detailed results.



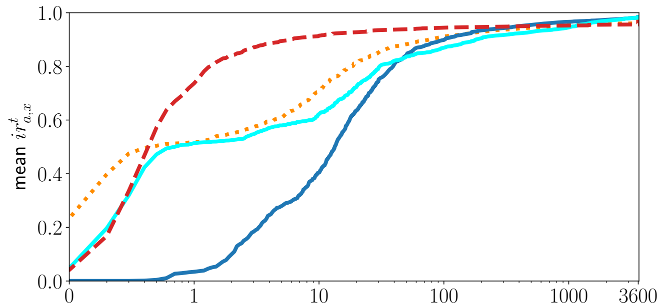
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

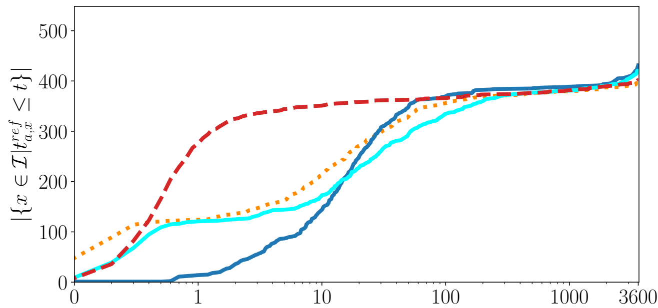
(c) Distribution of the maximum RAM value

Figure 6.9: Results of all algorithms for $Q|r_j, brkdwn|\sum T_j$

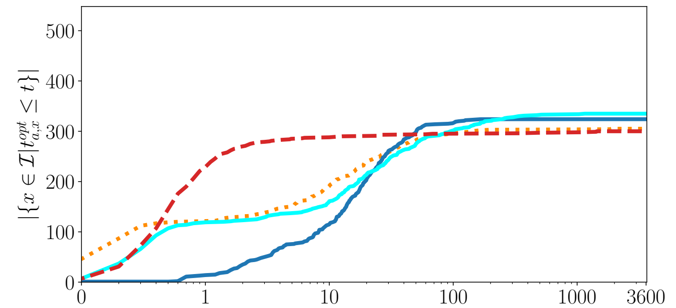
Detailed results analysis: Fig 6.10 shows the detailed results for the selected algorithms. It is important to notice that, contrary to previous cases, $CPO - ACO$ and CPO have similar performance after one hour. $CPO - ACO$ (resp. CPO) finds the reference solution on 420 instances (76.6% resp. 431, 78.6%) and prove optimality on 335 instances (resp. 324). Both methods have a mean inverse ratio equal to 0.98 after one hour. However, for short time limits, $CPO - ACO$ clearly outperforms CPO. For example, $CPO - ACO$ has a mean inverse ratio greater than 0.5 after 0.7 seconds, whereas CPO needs 13.4 seconds to have a mean inverse ratio greater than 0.5.
It is also important to highlight, once again, the good performance of $Tabu_S$ for short time limits. After 3 seconds, it finds the reference solution on 336 instances (61.3%, the second best approach after 3 seconds is $ACO - Tabu_J$ with 147 instances, 26.8%) and has a mean inverse ratio equal to 0.87 (against 0.57 for $ACO - Tabu_J$).
For really short time limits (lower than 0.5 seconds), $ACO - Tabu_J$ is the best method. However, for longer time limits it is always outperformed by $Tabu_I$.



(a) Evolution of average inverse ratio



(b) Time to reach reference solution



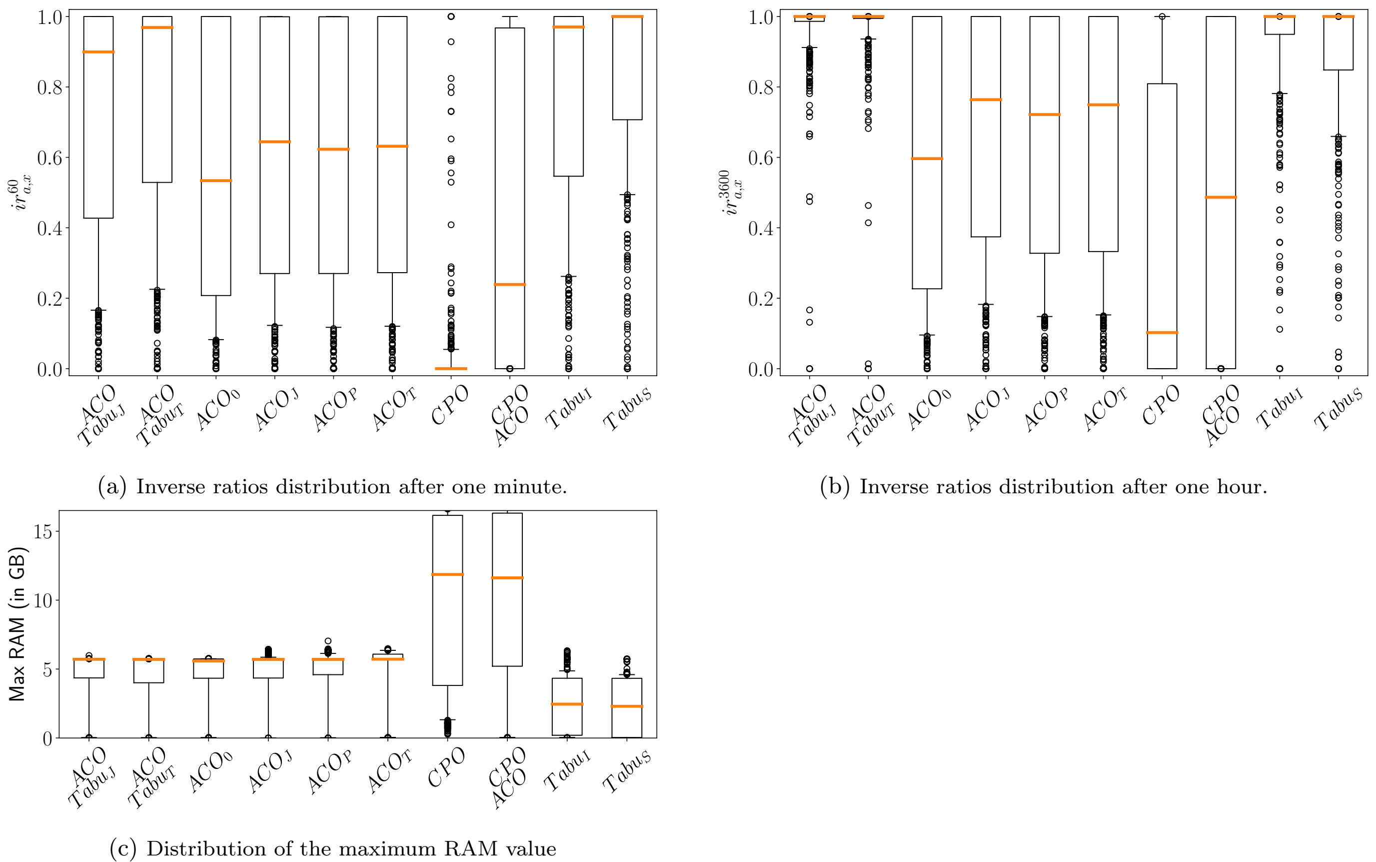
(c) Time to prove solutions optimality

Figure 6.10: Detailed results for $Q|r_j, brkdwn| \sum T_j$

6.2.6 Breaks and setup times: $Q|r_j, brkdwn, s_{jk}|\sum T_j$

Considered algorithms: The same algorithms as in the previous section are considered.

Global analysis: Fig 6.11 shows the results for all the considered algorithms. This section's main result is that CPO has much more difficulties to find good solutions than in previous problems. As aforementioned, the model is much more complicated than the ones in the previous sections (see section 2.5.5), and hence it is much more complicated for CPO to find good solutions. Furthermore, the consumed memory is higher than in the previous sections. For example, in the case with just the setup-times, the max memory consumed was 1.2GB of RAM on the worst instance, whereas for the problem dealt with in this section, CPO uses more than 11.5GB on more than 50% instances. Hence we do not consider CPO in the detailed results. Similarly to CPO, the complex CP model also penalizes $CPO - ACO$. Hence we do not consider $CPO - ACO$ neither in the detailed results.
As $Tabu_I$ is slightly better than $Tabu_S$, we only consider $Tabu_I$ in the detailed results. For a similar reason we do not consider $ACO - Tabu_J$, and none of the ACO variants as they are all outperformed by $ACO - Tabu_T$



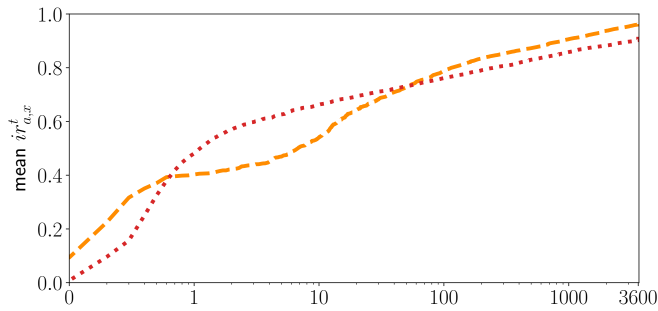
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

(c) Distribution of the maximum RAM value

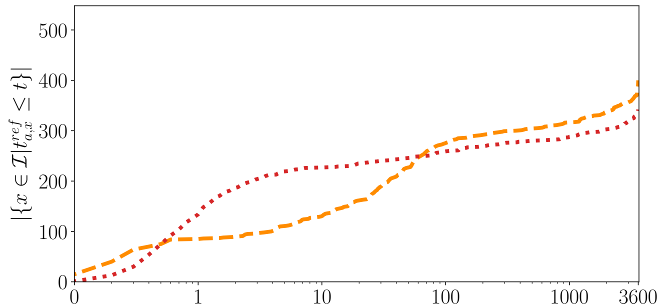
Figure 6.11: Results of all algorithms for $Q|r_j, brkdwn, s_{jk}|\sum T_j$

Detailed results analysis: Fig 6.12 shows the detailed results for the selected algorithms. The hybrid method $ACO - Tabu_T$ finds the reference solution on the greatest number of instances (400, 73.0%) and has the best mean inverse ratio after one hour (0.97). However, using $Tabu_I$ alone allows finding good solutions more quickly. After 3 seconds, $Tabu_I$ finds the reference solution on 204 instances against 95 instances for $ACO - Tabu_T$. $ACO - Tabu_T$ finds the reference solution on more instances for time limit greater than 98 seconds.

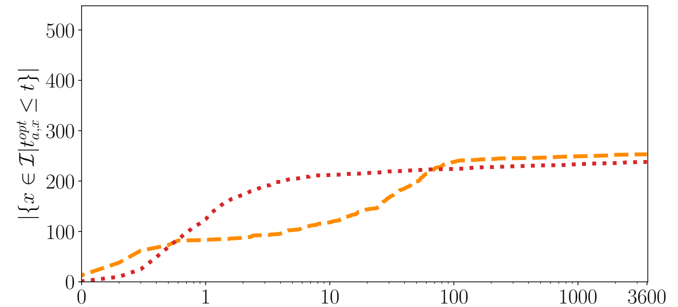


(a) Evolution of average inverse ratio

$ACO - Tabu_T$ $Tabu_I$



(b) Time to reach reference solution

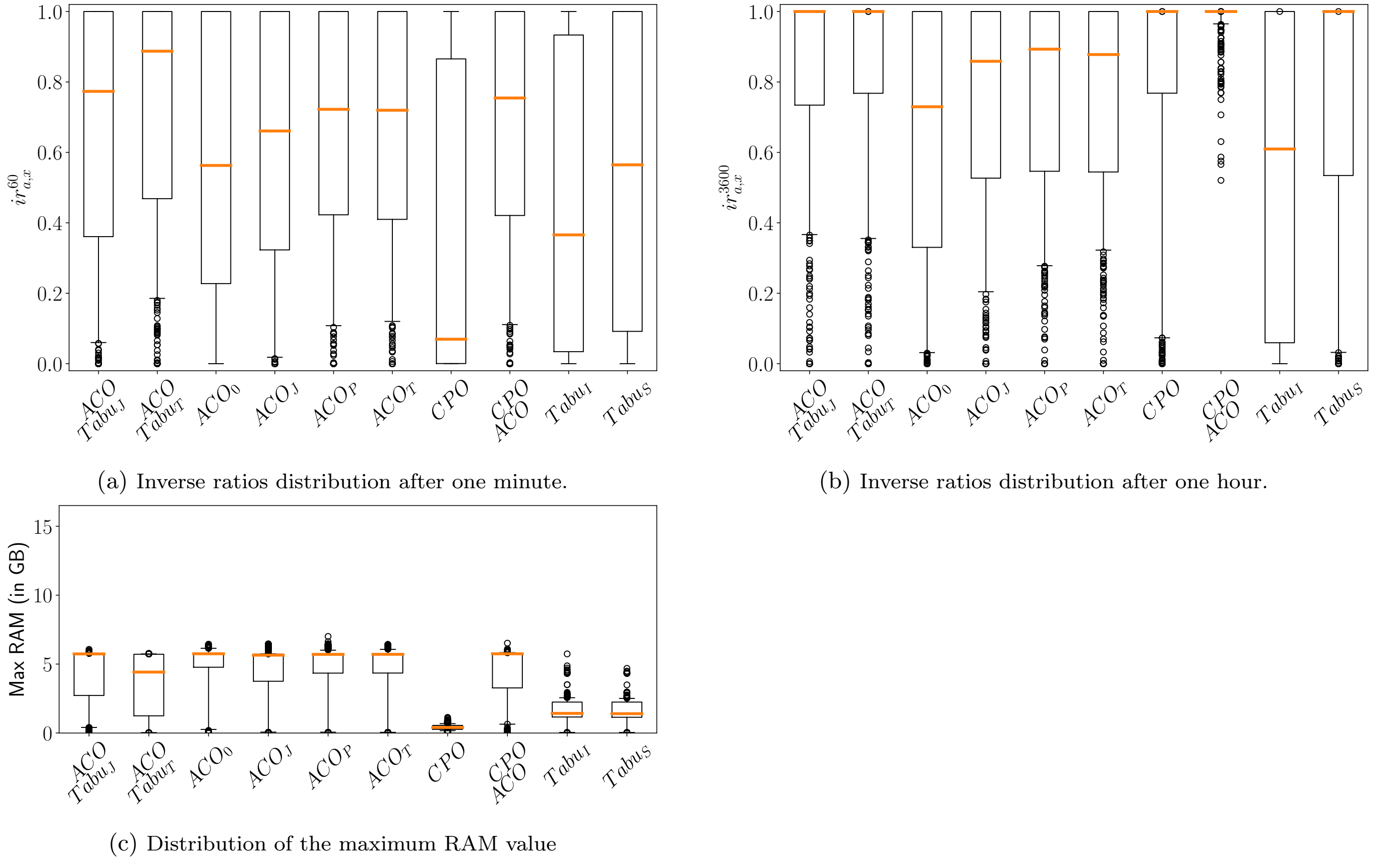


(c) Time to prove solutions optimality

Figure 6.12: Detailed results for $Q|r_j, brkdwn, s_{jk}| \sum T_j$

6.2.7 GC_{loose} no breaks: $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$

Considered algorithms: The same algorithms as in the previous section are considered.



(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

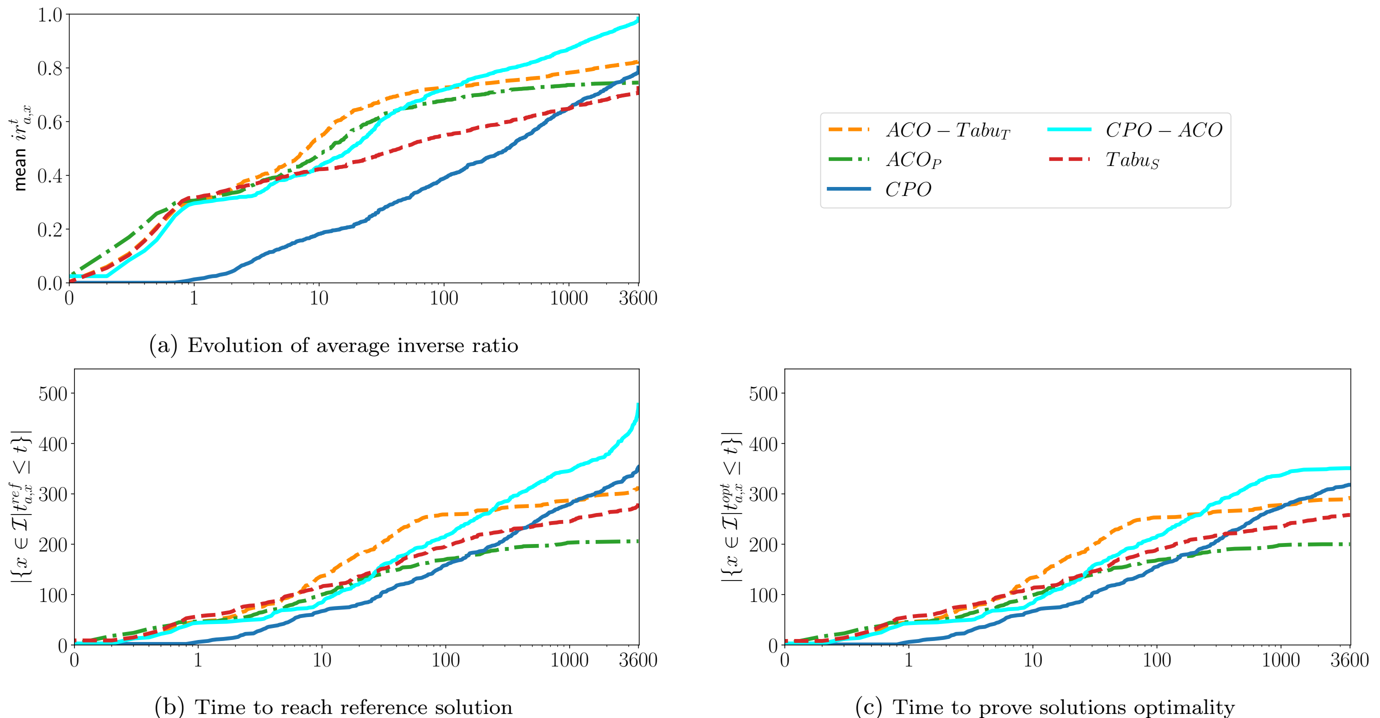
(c) Distribution of the maximum RAM value

Figure 6.13: Results of all algorithms for $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$

Global analysis: Fig 6.13 shows the results for all the considered algorithms. The most noticeable result is that $Tabu_I$ has great difficulties dealing with the GC constraint. This is a direct consequence of theorem 3.3.1. Indeed tabu search was efficient for problems listed in previous sections because it was trivial to convert a list-schedule into a real schedule. The method described in 4.2.1 makes tabu search find solutions, but it requires expensive computation to convert list-schedules into schedules. The median inverse ratio of $Tabu_I$ is equal to 0.60 after one hour. However, $Tabu_S$ has quite good results for this problem. Its median inverse ratio is equal to 1.0 after one hour. The fact that $Tabu_S$ has better results than $Tabu_I$ could be explained as follows: with $Tabu_I$ when a job j is removed from a machine and put on another machine, all the jobs that were after j can start earlier, and all the jobs which are after j on its new machine will probably start later in the new solution. Hence all the jobs which were after j in the previous solution or which are after j in the new solution have important changes in their start times. Hence all their groups have a huge probability of having their start or end times modified by this move, which results in a great probability of creating conflicts for GC (conflicts which are hard to solve). On the other hand, with $Tabu_S$, when we swap two jobs which have approximately the same durations, the jobs which are after them on their machines will probably not be modified by the move, and hence the probability of creating conflicts for GC is lower.
Another interesting point is that the gap between ACO_0 and other ACO variants is consequent. The median inverse ratio is equal to 0.73 (resp. 0.86, 0.89 and 0.88) for ACO_0 (resp. ACO_J, ACO_P and ACO_T). This clearly highlights the

positive role of the pheromone trails for this problem. As ACO_P outperforms other ACO variants, we only keep it for the detailed results.
For a similar reason, we do not show the detailed results of $ACO - Tabu_J$ as it is outperformed by $ACO - Tabu_T$.

Detailed results analysis: Fig 6.14 shows the detailed results for the selected algorithms. We remark that combining ACO and Tabu search works well when considering the GC constraint. As seen above, *tabu search* has great difficulties in computing many neighbors due to GC, and hence it is not efficient for long runs. However, it is efficient to improve a solution quickly when there are many improving neighbors. Thus it is useful when combined with ACO. Indeed, ACO generates many solutions, and tabu search can quickly improve them. As soon as tabu search begins to stagnate, it is stopped. Hence we minimize the time spent to convert list-schedules into schedules. Then these improved solutions help ACO generating new better solutions. After one hour it has a mean inverse ratio equal to 0.83 and it finds the reference solution on 315 instances (57.5%).
It is also noticeable that $CPO - ACO$ outperforms other approaches after one hour of computation. It finds the reference solution on 477 instances (87.0%) and proves optimality on 351 instances (64.1%). This approach is especially well-suited for this problem.
CPO alone also has good results as it finds the reference solution on 353 instances (64.4%) after one hour and has a mean inverse ratio equal to 0.80. However, for short time limits, it has worse results. $ACO - Tabu_J$ needs 1.1 seconds to reach a mean inverse ratio of 0.3 whereas CPO needs 41 seconds to reach such a mean inverse ratio.
We can also notice that ACO_P has better mean inverse ratio over time than $Tabu_S$ but it finds the reference solution on less instances (whichever the time limit). This shows that $Tabu_S$ is able to find the best solution on many instances, but ACO_P often finds better solutions than $Tabu_S$. In particular ACO_P is less sensitive to the input instance.



(a) Evolution of average inverse ratio

(b) Time to reach reference solution

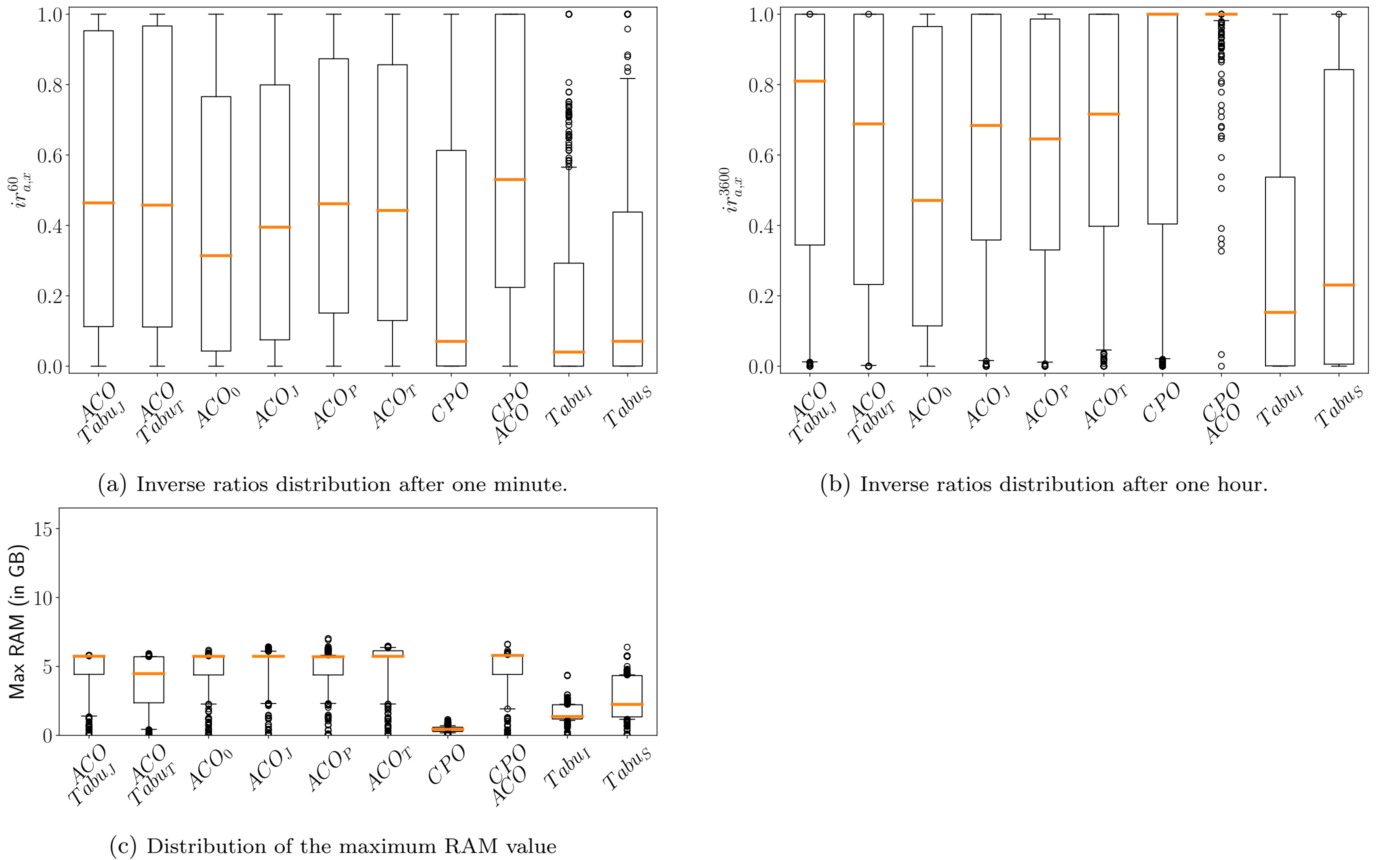
(c) Time to prove solutions optimality

Figure 6.14: Detailed results for $Q|r_j, s_{jk}, GC_{loose}| \sum T_g$

6.2.8 GC_{tight} no breaks: $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$

Considered algorithms: The same algorithms as in the previous section are considered.

Global analysis: Fig 6.15 shows the results for all the considered algorithms. For this problem, the conclusions found for the previous problem are even more highlighted. The difficulties encountered by $Tabu_I$ are more noticeable when the GC constraint is tighter. Furthermore, $Tabu_S$ also has bad results on this problem. Indeed with a tight GC the number of conflicts at each move in the neighborhood is great, and hence both $Tabu_I$ and $Tabu_S$ spend too much time in repairing solutions. For this reason, they are not considered in the detailed results.
We also notice that the gap between $CPO - ACO$ and other approaches is more consequent than in the previous section. Similarly to the previous case, the role of the pheromone is clearly highlighted. However, $ACO - Tabu_J$ outperforms all ACO variants and $ACO - Tabu_T$, and hence these methods are not displayed in the detailed results.



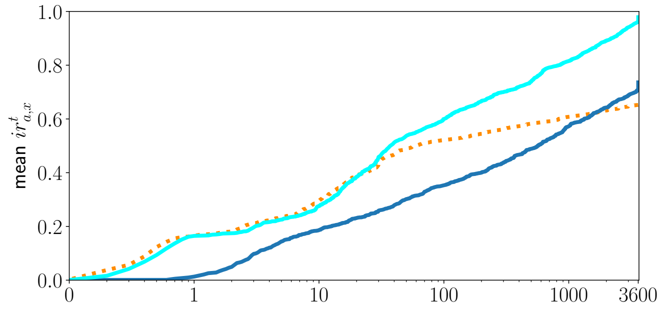
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

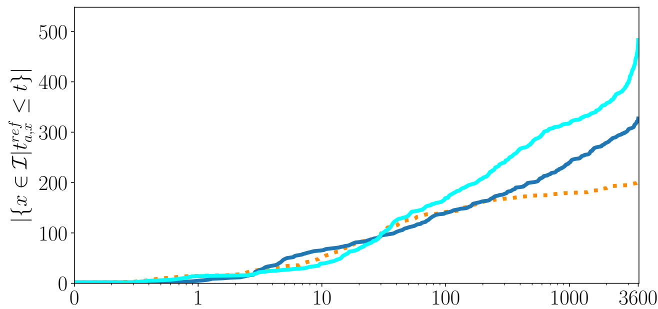
(c) Distribution of the maximum RAM value

Figure 6.15: Results of all algorithms for $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$

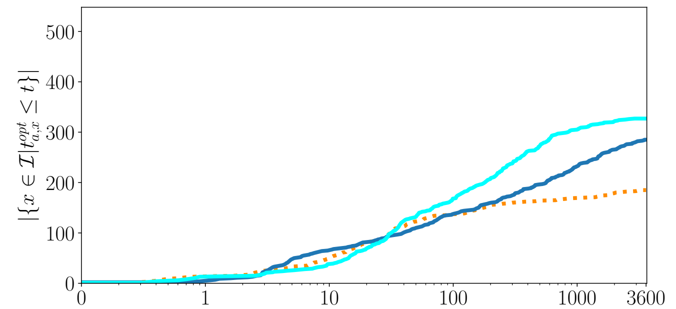
Detailed results analysis: Fig 6.16 shows the detailed results for the selected algorithms. As $Tabu_I$ has more difficulties, the combination of ACO and *tabu search* also has more difficulties. $ACO - Tabu_J$ finds the reference solution on 201 instances (36.7%), whereas it finds it on 317 instances (57.8%) when considering a loose GC.
We also observe that on this problem combining CPO and ACO is really efficient. It finds the reference solution on 483 instances (88.1%, whereas CPO alone finds it on 327 instances (59.7%)).



(a) Evolution of average inverse ratio



(b) Time to reach reference solution



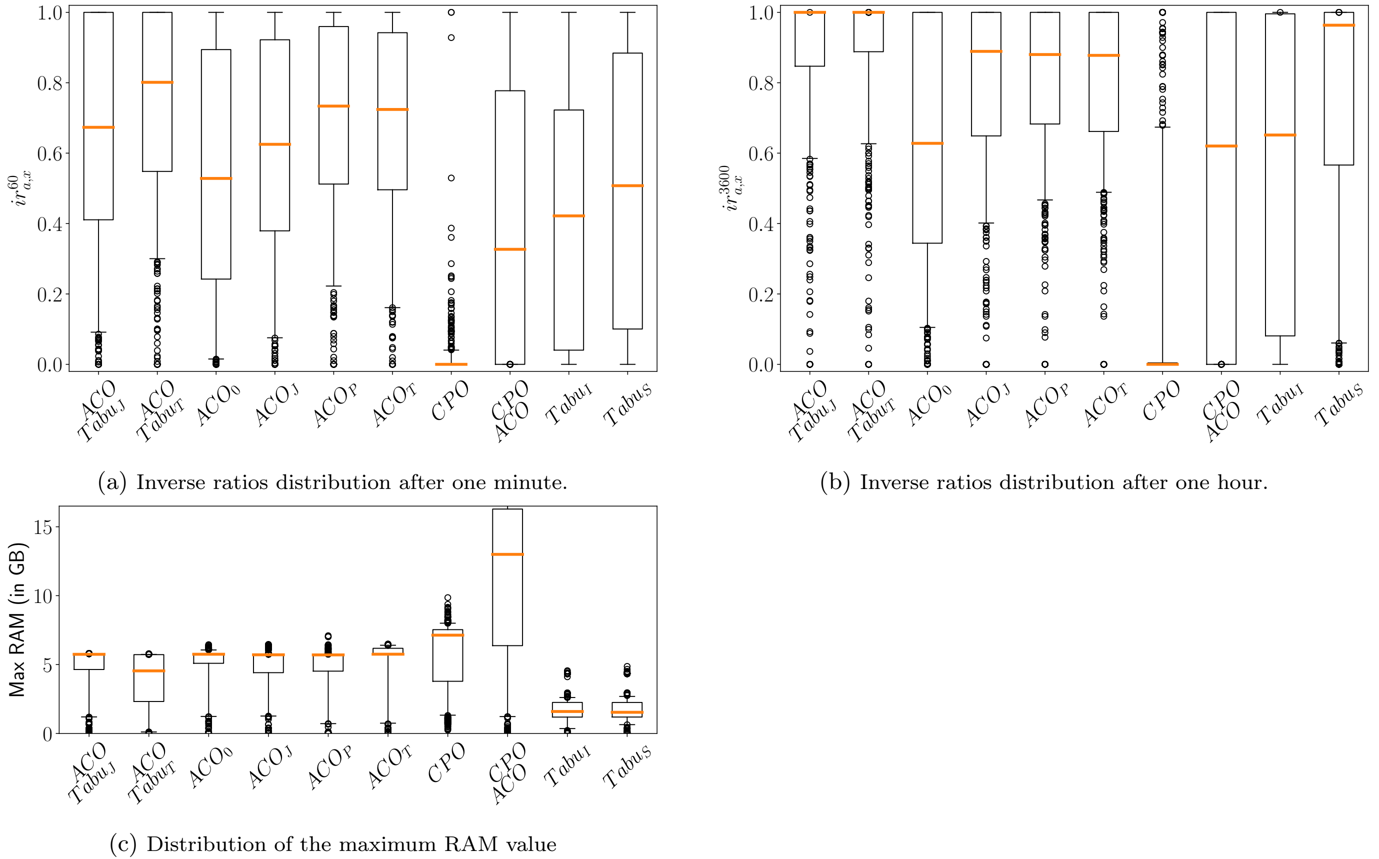
(c) Time to prove solutions optimality

Figure 6.16: Detailed results for $Q|r_j, s_{jk}, GC_{tight}| \sum T_g$

6.2.9 GC_{loose} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$

Considered algorithms: The same algorithms as in the previous section are considered.

Global analysis: Fig 6.17 shows the results for all the considered algorithms. For this problem, similarly to problem $Q|r_j, s_{jk}, brkdwn|\sum T_g$ which considers both scheduled breakdowns and setup-times, CPO has bad performance. This is due to the complex model used (see section 2.5.5). Hence combining CPO and ACO also leads to poor performance. Both approaches consume a lot of memory. Thus, these two approaches are not considered in the detailed results.
Once again, we clearly notice the interest of the pheromone on this problem: ACO_J, ACO_P and ACO_T have much higher median inverse ratios (0.89, 0.88, 0.88 respectively) than ACO_0 (0.63). As ACO_J has a better median ratio after one hour than the other ACO variants, we only consider it in the detailed results. We also notice that combining ACO and $Tabu$ is especially well-suited for this problem as both $ACO - Tabu_J$ and $ACO - Tabu_T$ have a median inverse ratio equal to one.
For the same reasons mentioned for problem $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$, the gap between the two $Tabu$ versions is also important.



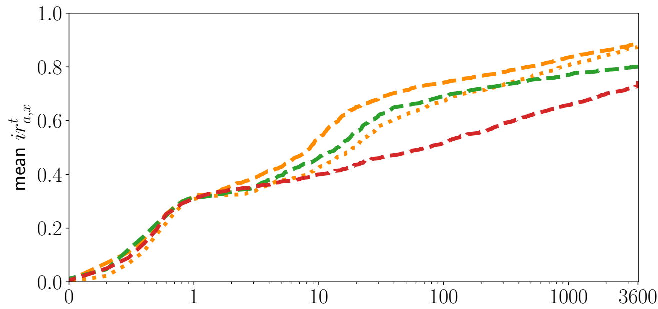
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

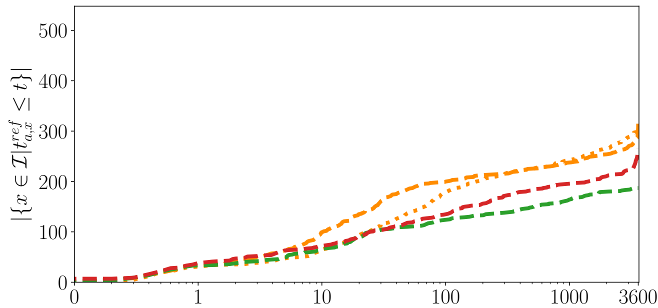
(c) Distribution of the maximum RAM value

Figure 6.17: Results of all algorithms for $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$

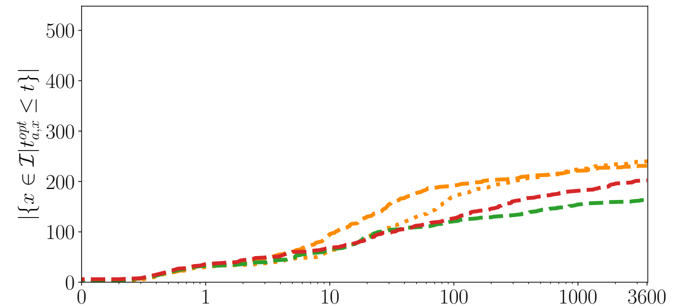
Detailed results analysis: Fig 6.18 shows the detailed results for the selected algorithms. Similarly to problem $Q|r_j, s_{jk}, GC_{loose}| \sum T_g$, combining ACO and $Tabu$ leads to good results ($ACO - Tabu_T$ finds reference solution on 321 instances (58.6%)).
We also notice that *time trails* have slightly better performance than *jobs trails* when combined with *tabu search*, especially for time limits in $[10, 100]$ seconds. As an example, $ACO - Tabu_T$ finds the reference solution on 150 instances in 27.5 seconds, whereas $ACO - Tabu_J$ needs 67 seconds to do so. This highlights the interest of the newly introduced pheromone structure.
Similarly to the previous cases considering GC, we observe that ACO has a good mean inverse ratio whereas it does not find the reference solution on many instances. Once again, this highlights the fact that it often finds solutions close to the best one, without reaching it.



(a) Evolution of average inverse ratio



(b) Time to reach reference solution



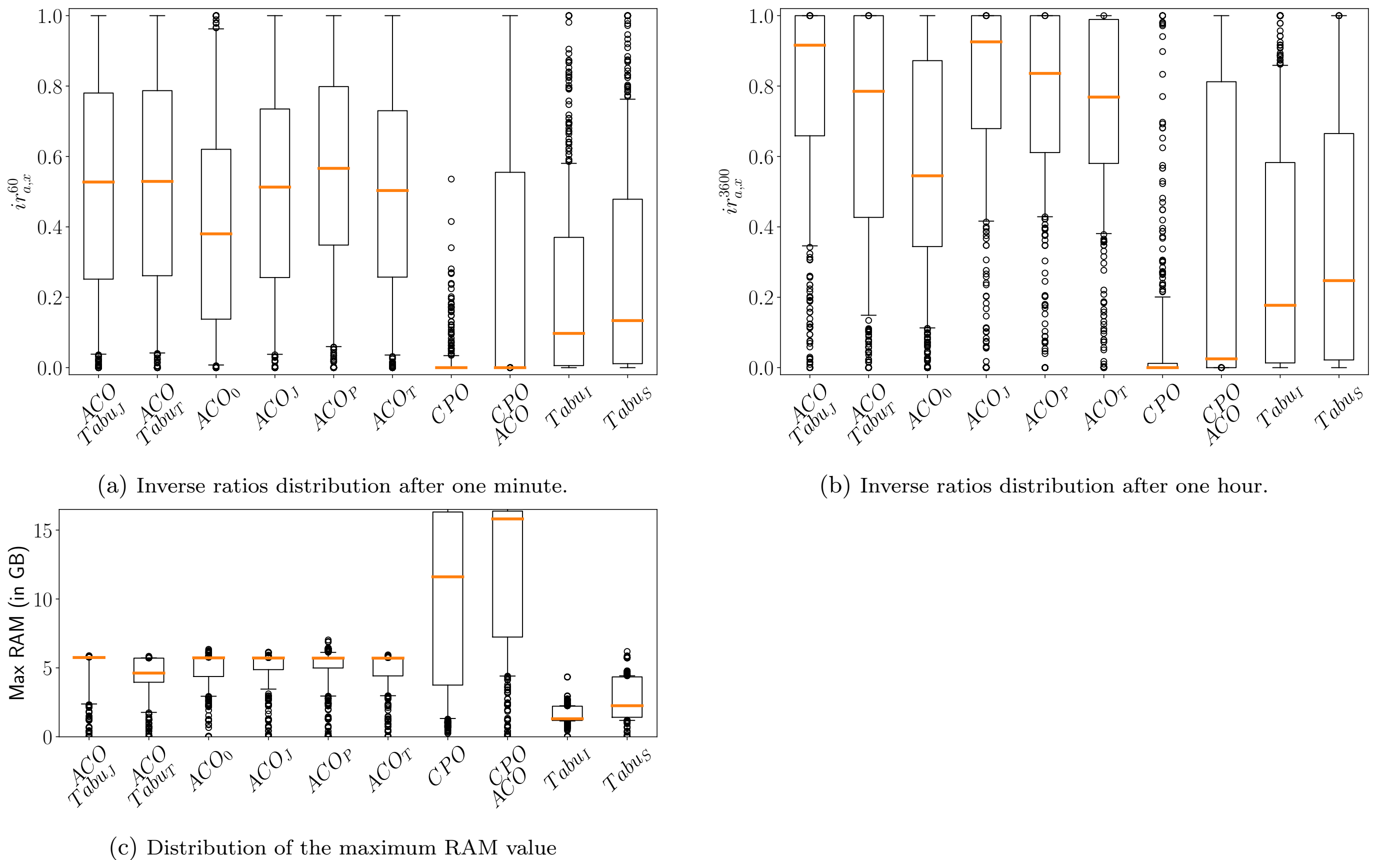
(c) Time to prove solutions optimality

Figure 6.18: Detailed results for $Q|r_j, s_{jk}, brkdwn, GC_{loose}| \sum T_g$

6.2.10 GC_{tight} with breaks: $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

Considered algorithms: The same algorithms as in the previous section are considered.

Global analysis: Fig 6.19 shows the results for all the considered algorithms. As expected, we notice that both $Tabu$ variants have difficulties for this problem, and that CPO and $CPO - ACO$ also have terrible performance for this problem. Hence none of these methods are considered in the detailed results.
We also notice, contrary to previous problems, that ACO_J has a median inverse ratio (0.93) better than the ones of both $ACO - Tabu$ methods (0.92 for $ACO - Tabu_J$ and 0.79 for $ACO - Tabu_T$). We also notice that the pheromone trails have different performance: ACO_j having the best results, and ACO_T the worst. As ACO_0 is worse than other methods, we do not consider it in the detailed results.



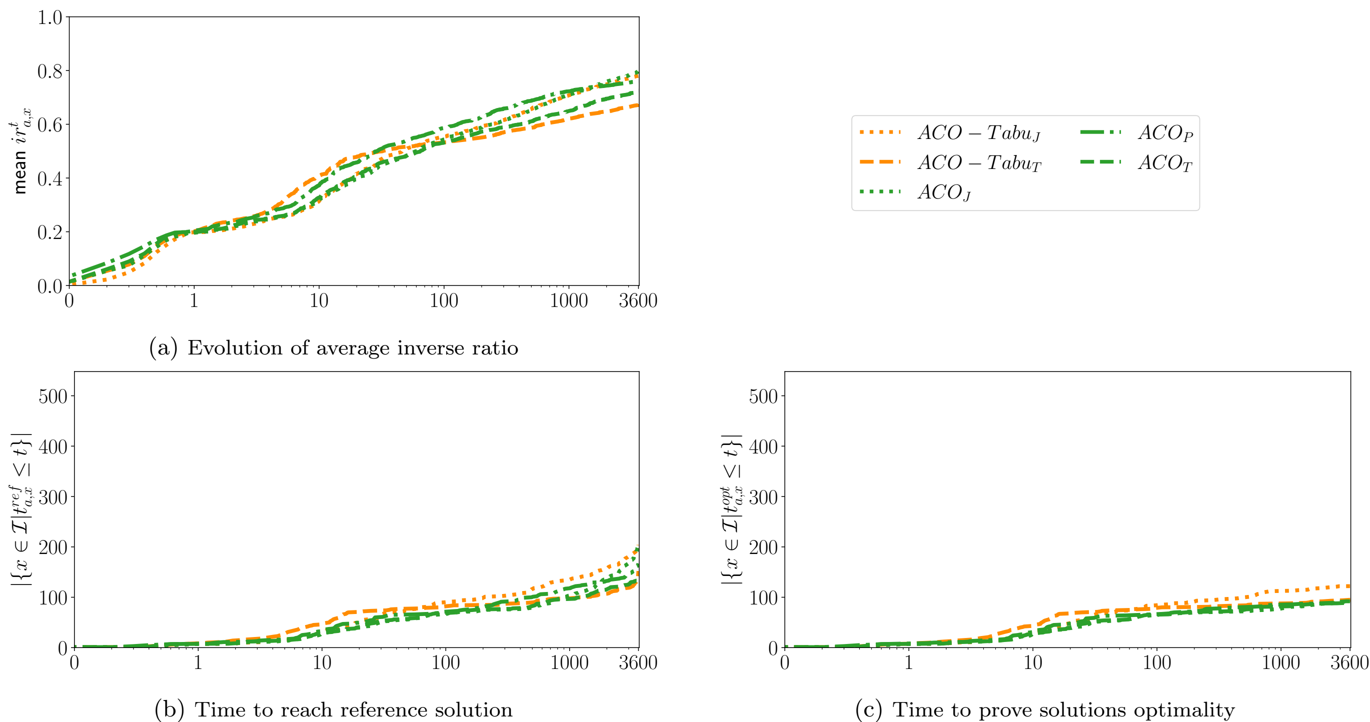
(a) Inverse ratios distribution after one minute.

(b) Inverse ratios distribution after one hour.

(c) Distribution of the maximum RAM value

Figure 6.19: Results of all algorithms for $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

Detailed results analysis: Fig 6.20 shows the detailed results for the selected algorithms. We notice that no method finds the reference solution on more than 204 instances (37.2%). This shows us that the different used algorithms have complementary strengths and weaknesses for this problem. We observe that according to the parameters (and hyper-parameters), *ACO* variants are able to solve different instances.
Another noticeable result is that combining *ACO* and *Tabu search* leads to results similar to the ones obtained using *ACO* alone. As noticed above, the tighter the *GC* constraint, the worst the results of *tabu search*. Hence the hybrid method also suffers of the difficulties encountered by *tabu search*.
It also shows that the newly introduced pheromone structure (denoted *Time*) has performance comparable with the two other classical pheromone structures, but the three pheromone structures work well on a different subset of instances.



(a) Evolution of average inverse ratio

(b) Time to reach reference solution

(c) Time to prove solutions optimality

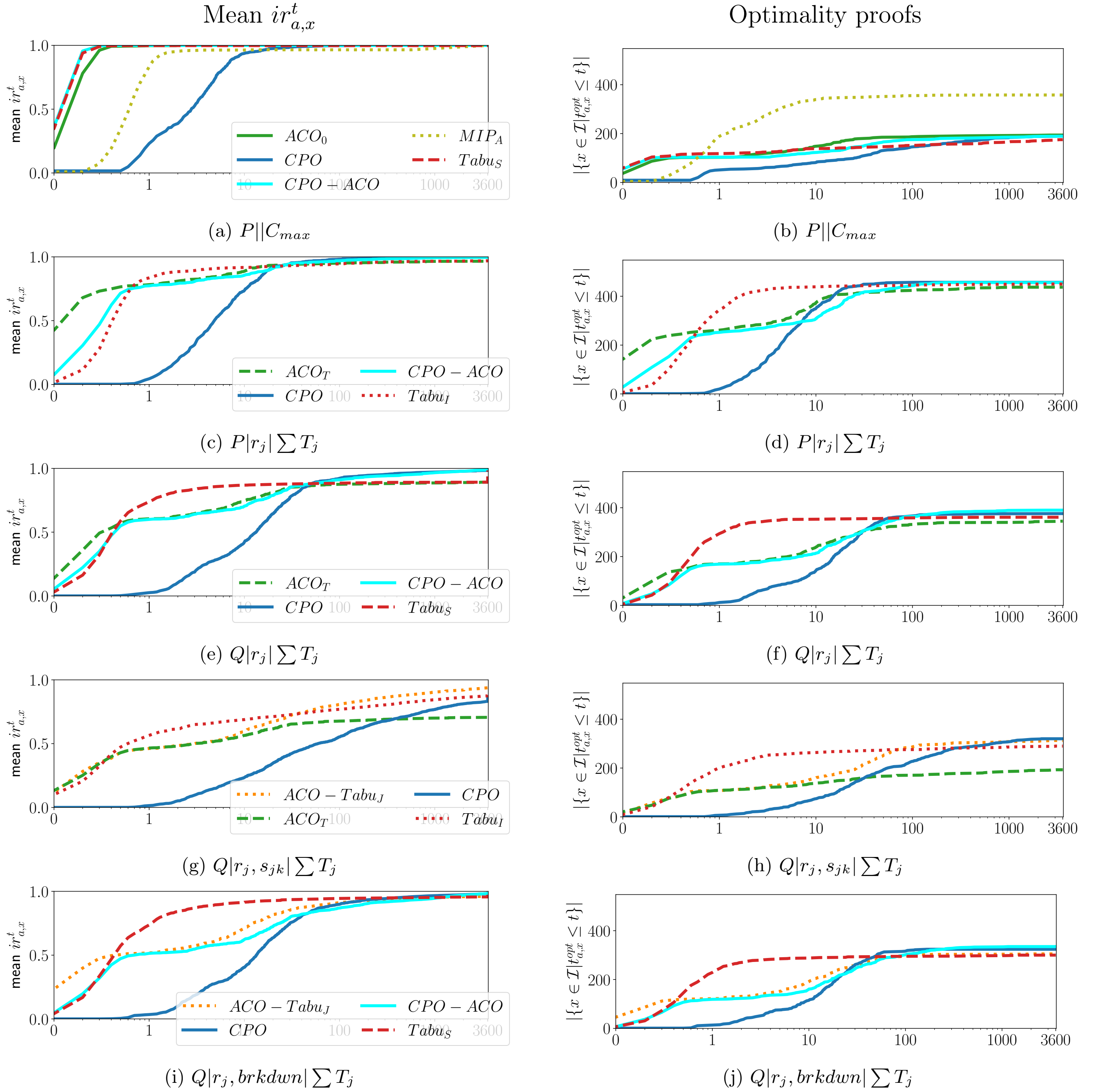
Figure 6.20: Detailed results for $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

6.3 Results over all problems

In Fig 6.21 and Fig 6.22 we show, for the different problems, the plot corresponding to the evolution of the mean inverse ratio with the reference solution and the evolution of the number of instances on which an optimality proof has been found. For each problem, we only plot results of the best performing approaches.

This figure shows us that:

- the best performing approaches are very different from a problem to another,
- the best performing approach often depends on the time limit considered,
- the approach that has the best average inverse ratio is not necessarily the approach that is able to prove optimality for the largest number of instances.



(a) $P||C_{max}$ (b) $P||C_{max}$

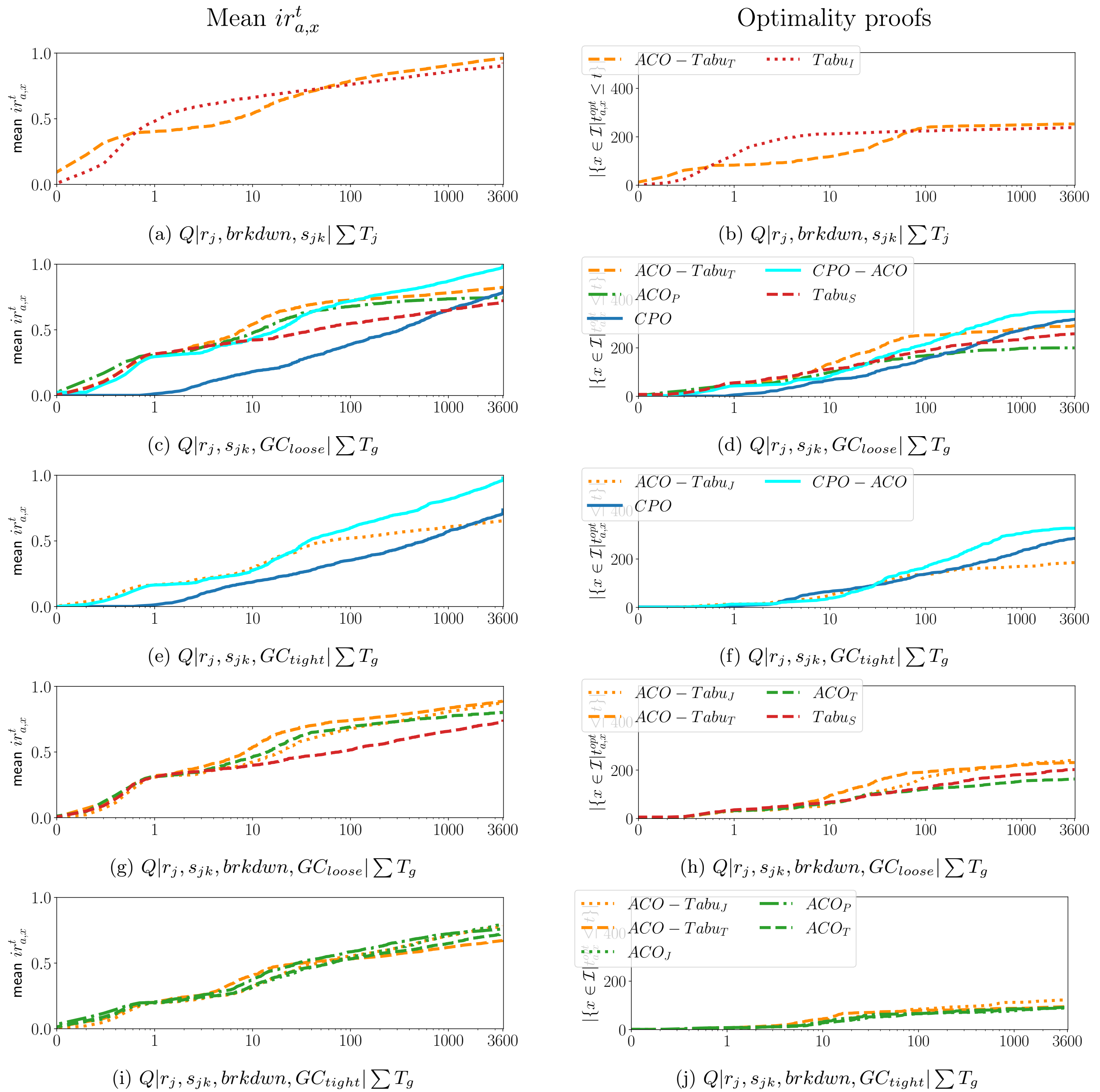
(c) $P|r_j|\sum T_j$ (d) $P|r_j|\sum T_j$

(e) $Q|r_j|\sum T_j$ (f) $Q|r_j|\sum T_j$

(g) $Q|r_j, s_{jk}|\sum T_j$ (h) $Q|r_j, s_{jk}|\sum T_j$

(i) $Q|r_j, brkdwn|\sum T_j$ (j) $Q|r_j, brkdwn|\sum T_j$

Figure 6.21: Evolution of the mean inverse ratio (left) and number of optimality proofs (right) for $P||C_{max}$, $P|r_j|\sum T_j$ and $Q|\beta|\sum T_j$ with $\beta \in \{\{r_j\}, \{r_j, s_{jk}\}, \{r_j, brkdwn\}\}$



(a) $Q|r_j, brkdwn, s_{jk}|\sum T_j$

(b) $Q|r_j, brkdwn, s_{jk}|\sum T_j$

(c) $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$

(d) $Q|r_j, s_{jk}, GC_{loose}|\sum T_g$

(e) $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$

(f) $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$

(g) $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$

(h) $Q|r_j, s_{jk}, brkdwn, GC_{loose}|\sum T_g$

(i) $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

(j) $Q|r_j, s_{jk}, brkdwn, GC_{tight}|\sum T_g$

Figure 6.22: Evolution of the mean inverse ratio (left) and number of optimality proofs (right) for $Q|r_j, brkdwn|\sum T_j$ and $Q|r_j, s_{jk}, \beta'|\sum T_g$ with $\beta' \in \{\{GC_{loose}\}, \{GC_{tight}\}, \{brkdwn, GC_{loose}\}, \{brkdwn, GC_{tight}\}\}$

	$ACO-Tabu_J$	$ACO-Tabu_T$	ACO_0	ACO_J	ACO_P	ACO_T	CPO	$CPO-ACO$	MIP_A	ORT	$Tabu_I$	$Tabu_S$
$P\|\|C_{max}$	513 93	519 **218**	547 51	$\perp$	$\perp$	$\perp$	541 1	538 73	**548** 194	56 0	233 189	483 162
$P\|r_j\|\sum T_j$	517 113	525 178	485 11	471 159	475 **205**	483 200	**538** 15	536 22	$\perp$	237 0	522 35	536 170
$Q\|r_j\|\sum T_j$	435 53	439 153	383 32	354 76	361 59	367 65	470 41	**476** 45	$\perp$	180 0	421 90	456 **190**
$Q\|r_j, s_{jk}\|\sum T_j$	414 117	399 110	203 2	202 24	201 45	206 52	383 30	**416** 47	$\perp$	0 0	362 141	361 **152**
$Q\|r_j, brkdwn\|\sum T_j$	396 105	398 111	317 33	306 58	302 48	304 44	**431** 64	420 44	$\perp$	$\perp$	362 79	402 **221**
$Q\|r_j, brkdwn,$ $s_{jk}\|\sum T_j$	371 73	**400** 161	167 9	165 30	165 22	166 33	81 1	160 30	$\perp$	$\perp$	342 124	314 **177**
$Q\|r_j, s_{jk},$ $GC_{loose}\|\sum T_g$	317 20	315 104	184 5	201 18	206 52	208 42	353 84	**477** **142**	$\perp$	$\perp$	166 46	281 86
$Q\|r_j, s_{jk},$ $GC_{tight}\|\sum T_g$	201 33	179 77	119 17	138 7	130 39	149 14	327 136	**483** **218**	$\perp$	$\perp$	58 7	123 27
$Q\|r_j, s_{jk}, brkdwn,$ $GC_{loose}\|\sum T_g$	307 84	**321** **169**	146 9	182 30	182 70	187 42	26 3	143 7	$\perp$	$\perp$	136 52	265 103
$Q\|r_j, s_{jk}, brkdwn,$ $GC_{tight}\|\sum T_g$	**204** **109**	152 103	97 17	201 **109**	167 98	133 50	6 1	88 14	$\perp$	$\perp$	33 4	75 46
Mean values	367.5 80.0	364.7 **138.4**	264.8 18.6	276.7 56.2	273.6 68.9	275.0 59.3	315.6 37.6	**373.7** 64.2	$\perp$	$\perp$	263.5 76.7	329.6 133.4

Figure 6.23: Number of instances on which $t_{a,x}^{ref} < 3600$ (top of line) and number of instances on which $t_{a,x}^{ref}$ is minimal (among all algorithm for the considered instance). The best value of each line is bold.

In Fig. 6.23 we display for each problem and each algorithm a the number of instances on which $t_{a,x}^{ref} < 3600$ (top of line) and the number of instances on which $t_{a,x}^{ref}$ is minimal (among all algorithm for the considered instance). The best value of each line is bold and green. It is important to notice that the bold values are not concentrated in the same column, proving that, depending on the considered constraints, the best approach vary. On the last line of the table, we display the mean values over all the problems.[2]

6.4 Discussion

In this chapter we evaluate a set of state-of-the-art algorithms on a new benchmark of scheduling problems. In particular the same instances are used but with different constraints. This allows us to evaluate the influence of the different constraints on the methods.
As a conclusion we can notice that *tabu search* is especially efficient when considering scheduling problems without *GC* constraints. It finds good solutions in small amount of time. However, when adding a *GC* constraint, it has much more difficulties to solve problems. This a direct consequence of theorem 3.3.1.
ACO is able to quickly find good solutions on scheduling problems with few constraints, but their quality after one hour of computation is generally worse than the quality of the solutions found by *tabu search* or *CPO*. However, as more constraints are considered, *ACO* becomes more competitive with other methods. In particular, when considering setup-times, breakdowns and *GC*, variants of ACO are the only methods which are able to find good solutions.
We observe that *Linear Programming* has great difficulties as soon as start and end times must be found for each job on our data-set. It is especially well-suited for assignment problems, but it has much more difficulties to deal with the scheduling problems on our data-set.
CP Optimizer is really well-suited for scheduling problems. It usually finds good solutions after one hour of computation. The only exception lies in scheduling problems considering both setup-times and breakdowns. In that case, as the model is much more complicated (see section 2.5.5), it is less competitive with other approaches.
It is also worth mentioning that combining approaches often leads to taking the best of both worlds. For example combining *ACO* and *Tabu search* often leads to results which are better than considering *ACO* alone or *tabu search* alone. This is also the case when considering the newly introduced algorithm *CPO-ACO* which often combines the speed of ACO with the quality of CPO. The results are especially outstanding for problems combining setup-times and *GC* constraints.

As explained in chapter 5, Copilote ERP faces different scheduling problems and must be able to solve them as best as possible. However, as seen in this chapter, according to the problem, the method which leads to the best results differs. Hence, in the next chapter we will apply *automatic selection algorithm* to automatically find the best algorithm (along with its parameters) for each instance according to its features.

[2] For algorithms ACO_J, ACO_P and ACO_T and for problem $P||C_{max}$ the values obtained by ACO_0 have been selected to compute the mean.

Chapter 7

Automatic Algorithm Selection

Contents

7.1 Definition and goals of automatic algorithm selection 123
7.2 Performance measure 124
7.3 Instance features 125
7.4 Choice of an Approach for Learning the Mapping Model 127
7.5 Llama results 128
7.6 Discussion 130

As seen in the previous chapter, the best algorithm varies depending on the scheduling problem. Moreover, given a single algorithm, its performance on a given instance highly depends on its parameters (and hyper-parameters). Hence choosing a single algorithm (with a single parameter configuration) for all instances would prevent from obtaining the best overall performance. As explained in chapter 5 our goal is to develop a method able to solve the different scheduling problems faced by the *Copilote* ERP.
In this chapter we give a proof of concept about automatic algorithm selection. In section 7.1 we present automatic algorithm selection and we give a short survey of the existing work concerning this problem. In sections 7.2, 7.3 and 7.4 we give the context and the settings we use to apply automatic algorithm selection for our scheduling problems. Finally in section 7.5 we show some results of *Automatic algorithm Selection* in our context.

7.1 Definition and goals of automatic algorithm selection

Our goal is to select the best solving approach for each instance of each scheduling problem. A first possibility to reach this goal is to select, for each kind of scheduling problem, a good approach that performs well on all instances of the problem. This selection may be done manually (as proposed in [MPE03]], for example) or automatically by using automatic algorithm configuration tools [HHLS09, AST09, BYBS10, HHL11]. These tools search for the best (hyper)parameter setting of a parametrised algorithm given a set of training instances, assuming that the best setting for the training instances will also perform well on other instances of the problem. However, experimental results reported in the previous chapter show us that an approach that performs very well on some instances may have very poor performance on other instances of the same problem. Hence, we cannot use this kind of approach in our context, and we propose to use per-instance algorithm selection.

The problem of *algorithm selection* has been introduced by [Ric76]. The formulation of the problem can be stated as follows:

Definition 7.1.1 (Algorithm selection problem)**.**
Given:

- a set $\mathcal{I}$ of problem instances drawn from a distribution $\mathcal{D}$
- a set of algorithms $\mathcal{A}$
- a performance measure $m : \mathcal{I} \times \mathcal{A} \rightarrow \mathbb{R}$

the *per-instance algorithm selection problem* is to find a mapping $s : \mathcal{I} \rightarrow A$ that optimizes $E_{i \sim D}\ m(i, s(i))$, i.e., the expected performance measure for instances i distributed according to D. The mapping s is usually learned through an offline training step. Then, each time a new instance i must be solved, we use s to select the algorithm $s(i)$ that is expected to perform best on i.

Since its introduction this problem has received a lot of attention [BKK+16, Kot16]. It has been successfully applied for different problems: for example for the SAT problem [XHHL08, NMJ09], or for Algebraic Systems [DDE+05] or Multi-Mode Resource Constrained Project Scheduling Problem (MRCPSP) [MDC14]. More recently, algorithm selection has also been applied to continuous problems. The article [KHNT18] gives a good survey for both discrete and continuous cases.

The offline training of the mapping model $s : \mathcal{I} \rightarrow \mathcal{A}$ is a classification problem, and we can use classical machine learning algorithms (such as *random forests*, *KNN*, *neural networks*,... for example) to solve it. These algorithms are usually defined for classifying objects described by numerical vectors. Hence, we must define a procedure for extracting a numerical vector from an instance.

In this chapter, we use the R package Llama [Kot14] to learn the mapping model.

This preliminary work aims at giving a proof of concept that automatic algorithm selection can help in designing an approach that can achieve good performance whatever the considered instance. In that context we only consider the $Q|r_j, brkdwn, s_{jk}, GC_{tight}| \sum T_g$ problem and show that we can improve results by selecting algorithms on a per-instance basis.

7.2 Performance measure

As stated in Def. 7.1.1, the algorithm selection problem uses a performance measure $m : \mathcal{I} \times \mathcal{A} \rightarrow \mathbb{R}$ to evaluate the performance of an algorithm on an instance. A classical performance measure is the time needed to solve an instance. However, in the case of $\mathcal{NP} - hard$ problems, some instances cannot be solved within a reasonable amount of time and we must introduce a time limit (*i.e.*, one hour in our study). Furthermore, we need to define what we mean by "solving an instance". We may consider that an instance is solved when the optimal solution has been found (and the optimality proof has been done). However, there are a lot of instances that are not solved to optimality. Hence, we consider that an instance is solved when the reference solution has been found within the time limit.

Let us define a first performance measure, denoted m_{01}, that simply evaluates whether an instance has been solved or not:

$$\forall i \in \mathcal{I}, \forall a \in \mathcal{A}, m_{01}(i,a) = \begin{cases} 0 & \text{if } a \text{ finds the reference solution on } i \text{ within one hour} \\ 1 & \text{otherwise} \end{cases}$$

When using this measure, the selection model aims at maximising the expected number of solved instances within one hour.

This first measure does not consider the time needed to solve an instance. Let us consider the case where two algorithms a_1 and a_2 both solve an instance i, but a_1 is much faster than a_2. In this case, $m_{01}(i, a_1) = m_{01}(i, a_2)$ and the selection model has no reason to prefer a_1 to a_2.

Hence, we define a second performance measure, denoted m_t, to take into account the solving time:

$$\forall i \in \mathcal{I},\ \forall a \in \mathcal{A}\ m_t(i,a) = \begin{cases} t^{ref}_{a,i} & \text{if } t^{ref}_{a,i} < \infty \\ 3600/\min(ir^{3600}_{a,i}, 0.1) & \text{otherwise} \end{cases}$$

where $t_{a_i}^{ref}$ and $ir_{a_i}^{3600}$ are performance measures defined in Section 6.1.2. When an instance is not solved, $m_t(i,a)$ is greater than 3600, and the farer from the reference solution, the greater $m_t(i_a)$. As $ir_{a_i}^{3600}$ may be equal to 0, we lower bound the ratio with 0.1.

When using this second measure, the selection model aims at minimizing the expected time needed to find a good solution. For example, if $t_{a_1,i}^{ref} = 100$, $t_{a_2,i}^{ref} = 1000$, $t_{a_3,i}^{ref} = \infty$ et $ir_{a_3,i} = 0.5$ (*i.e.* the best solution found by a_3 is twice as large as the reference solution). In this case $m_t(i,a_1) = 100$, $m_t(i,a_2) = 1000$, and $m_t(i,a_3) = 3600/0.5 = 7200$,

Fig. 7.1 shows the results for this two different measures on the considered problem. We display the results of the *virtual best solver* (denoted VBS_m with m the considered measure in $\{m_{01}, m_t\}$). The *virtual best solver* is defined as a solver that perfectly selects the best approach from $\mathcal{A}$ on a per-instance basis; it provides a lower bound on the performance of any realistically achievable algorithm selector. More precisely, given a performance measure m in $\{m_{01}, m_t\}$, the performance of the VBS on an instance i is equal to $\min_{a \in \mathcal{A}} m(i,a)$.

By considering Fig. 7.1 we notice that VBS_t finds more quickly the reference solution than VBS_{01}. We also observe that between 10 and 1000 seconds, VBS_t finds better solutions than VBS_{01}. In particular, VBS_t reaches a value of mean $ir_{ax}^t = 0.5$ for $t = 27$ seconds whereas VBS_{01} needs 46 seconds to reach this value. Similarly, VBS_t finds the reference solution on 100 instances in 51 seconds whereas VBS_{01} needs 154 seconds to find the reference solution on 100 instances.

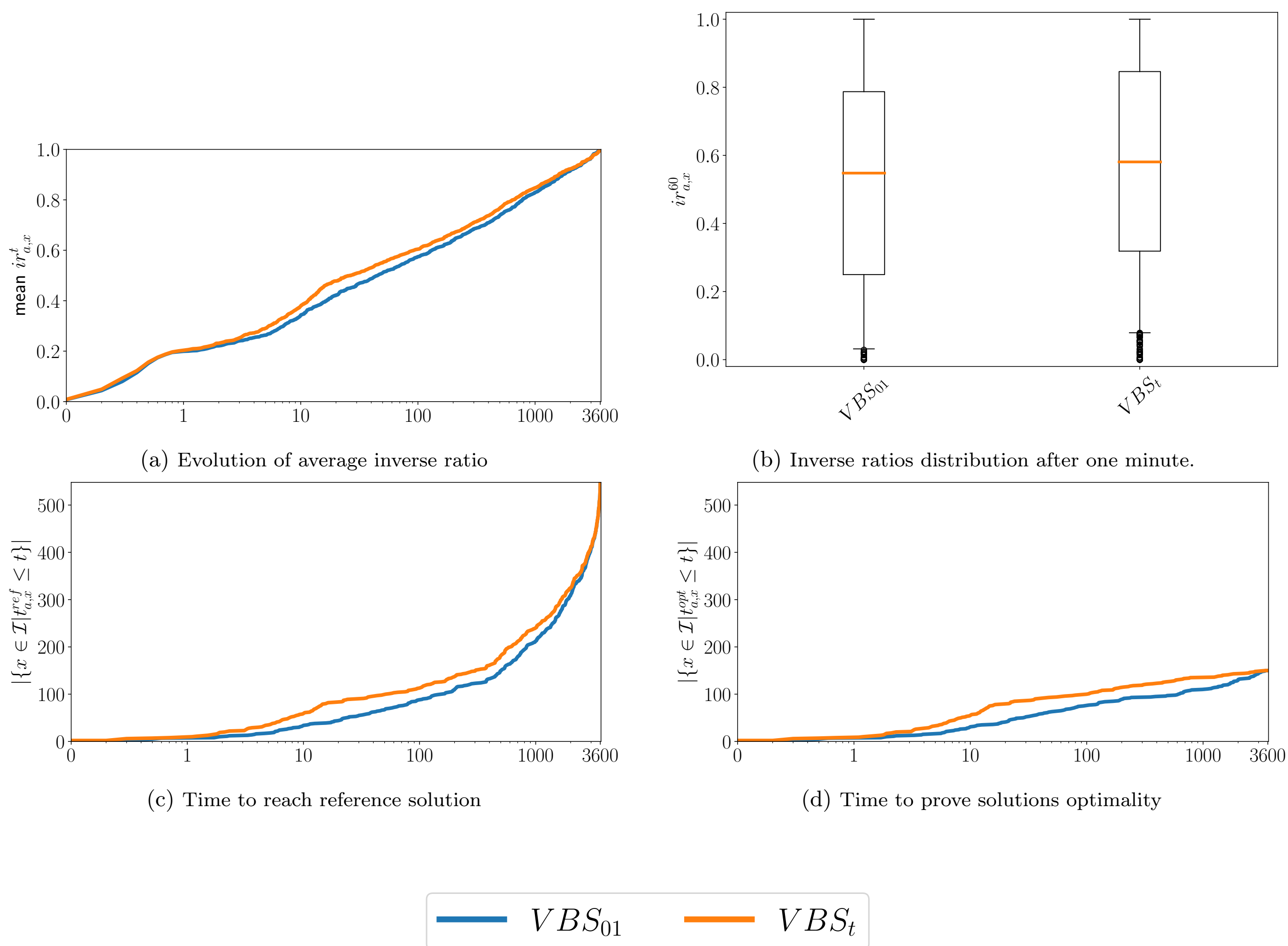
7.3 Instance features

In order to use Llama, we must describe each instance by a numerical vector. Given a sample X of n values, $stat(X)$ denotes the tuple that contains the four following values:

- the mean value $\overline{x} = 1/n * \sum_{x \in X} x$
- the minimum value in X
- the maximum value in X
- the variance $1/n \sum_{x \in X} (x - \overline{x})^2$ where $\overline{x}$ is the mean value of X.

An instance of $Q|r_j, brkdwn, s_{jk}, GC_{tight}| \sum T_g$ is described by the following features:

- The number of jobs n
- The number of machines m
- The number of groups $|\mathcal{P}|$
- The value of the limit l in the GC constraint
- Statistics on the job durations $stat(\{p_j | j \in \mathcal{J}\})$
- Statistics on the release dates $stat(\{r_j | j \in \mathcal{J}\})$
- Statistics on the due-dates $stat(\{d_j | j \in \mathcal{J}\})$
- Statistics on the sequence-dependent setup-times $stat(\{s_{j,k} | \{j,k\} \subseteq \mathcal{J}\})$
- The due date tightness as defined in [LPT97]: $1 - \overline{d}/C_{max}$ where $\overline{d}$ is the average of the due-dates and C_{max} is an approximation of the makespan (see [LPT97] to estimate C_{max})
- The due date range factor: $(d_{max} - d_{min})/C_{max}$
- The setup-time severity: $\overline{s}/\overline{p}$ where $\overline{s}$ is the average of the setup-times and $\overline{p}$ is the average of the processing-times.
- The job machine factor: n/m



(a) Evolution of average inverse ratio

(b) Inverse ratios distribution after one minute.

(c) Time to reach reference solution

(d) Time to prove solutions optimality

Figure 7.1: Results for the two different performance measures m_{01} and m_t

- Statistics on the number of jobs per group $stat(\{|g| : g \in \mathcal{P}\})$
- Statistics on the mean job duration in groups $stat(\{1/|g|\sum_{j\in g} p_j | g \in \mathcal{P}\})$
- Statistics on the minimum job duration in groups $stat(\{\min_{j\in g} p_j | g \in \mathcal{P}\})$
- Statistics on the maximum job duration in groups $stat(\{\max_{j\in g} p_j | g \in \mathcal{P}\})$
- Statistics on the variance of job durations in groups $stat(\{1/|g|\sum_{j\in g}(p_j - \overline{p_g})^2 | g \in \mathcal{P}\})$, where given a group $g \in \mathcal{P}$ $\overline{p_g}$ is the mean duration of its jobs ($i.e.$, $\overline{p_g} = 1/|g|\sum_{j\in g} p_j$)
- Statistics on the number of breaks per machine $stat(\{|\mathcal{B}_i| | i \in \mathcal{M}\})$, where given a machine $i \in \mathcal{M}$, $\mathcal{B}_i$ is the set of its breaks, $i.e.$, $\mathcal{B}_i = \{[e^i_l, b^i_{l+1}] | l \in [1, |\mathcal{A}_i|]\}$ where $\mathcal{A}_i, b^i_l$ and e^i_l are defined in Section 1.2.5.
- Statistics on the mean breaks duration per machine $stat(\{1/(|\mathcal{B}_i|)\sum_{B\in\mathcal{B}_i} |B| | i \in \mathcal{M}\})$, where given an interval $B \in \mathcal{B}_i$, $|B|$ represents its duration ($i.e.$, if $B = [e, b]$ then $|B| = b - e$).
- Statistics on the min break duration per machine $stat(\{\min_{B\in\mathcal{B}_i} |B| | i \in \mathcal{M}\})$
- Statistics on the max break duration per machine $stat(\{\max_{B\in\mathcal{B}_i} |B| | i \in \mathcal{M}\})$
- Statistics on the variance of break durations per machine $stat(\{1/|\mathcal{B}_i|\sum_{B\in\mathcal{B}_i}(|B| - \overline{B_i})^2 | i \in \mathcal{M}\})$, where given a machine $i \in \mathcal{M}$ $\overline{B_i}$ is the mean duration of its breaks ($\overline{B_i} = 1/(|\mathcal{B}_i|)\sum_{B\in\mathcal{B}_i} |B|$)

Finally we remove the quantities which are identical for all the instances. For example, the min over all the groups of the min job duration in the group is always equal to the min job duration over all the jobs.

7.4 Choice of an Approach for Learning the Mapping Model

The R package Llama [Kot14] offers the possibility to connect many algorithm selection approaches. It enables access to classification, regression, and clustering models for algorithm selection. In particular, it is possible to connect it to the *mlr* R package [BLK+16], which itself acts as an interface to the machine learning models provided by other R packages. In table 7.2a we report the sum of the misclassification penalties over all the instances for the performance measure m_{01}. The misclassification penalty mp_m for a given instance i and a given solver s is the difference of the values of the performance measure m between the algorithm chosen by the mapping model s and the algorithm chosen by the VBS: $mp_m(i, s) = m(i, s(i)) - m(i, VBS(i))$ where $VBS(i) = \underset{a\in\mathcal{A}}{argmin}\ m(i, a)$. In particular the misclassification penalty of the VBS is always zero. Besides, for the performance measure m_{01}, as the VBS finds for each instance an algorithm which reaches the reference solution, we have $\forall i \in \mathcal{I}\ m_{01}(i, VBS(i)) = 0$. So

$$mp_{m_{01}}(i, s) = m_{01}(i, s(i)) = \begin{cases} 0 & \text{if } s(i) \text{ finds the reference solution on } i \text{ within one hour} \\ 1 & \text{otherwise} \end{cases}$$

Hence the sum of the misclassification penalties over all the instances is just the number of instances on which the solver selected by s is unable to find the reference solution within one hour.

It is interesting to display the misclassification penalties for the measure m_{01} because it is easily interpretable. Hence table 7.2a shows that the learners' performance on the considered problem varies significantly. It is essential to notice that each learner uses randomness, and so the displayed results for each learner corresponds to the mean value obtained over four runs. We have tested over 50 learners and the results of only a subpart is displayed. We only display the solvers

Name	Misclass.	Name	Misclass.	Name	Misclass.
regr. random Forest	325.25	classif.earth	323.75	regr. Liblinea RL2L2SVR	323.25
regr. extraTrees	322.25	regr.ranger	321.25	classif. Liblinea RL1LogReg	320.50
regr.rsm	320.00	regr. random Forest	318.75	regr.cforest	318.50
regr.gbm	317.50	regr.RRF	317.50		

(a) Sum of misclassification penalties for different learners (for m_{01})

Name	Misclas.	Name	Misclas.	Name	Misclas.
regr.RRF	3062.81	regr. Liblinea RL2L2SVR	3034.11	regr.cforest	3019.46
regr. extraTrees	3001.38	regr. random Forest	2997.00	regr.gbm	2985.81
regr.ranger	2982.48	regr. random ForestSRC	2980.83	classif.earth	2973.38
classif. Liblinea RL1LogReg	2901.99	regr.rsm	2875.22		

(b) Mean misclassification penalty for different learners (for m_t)

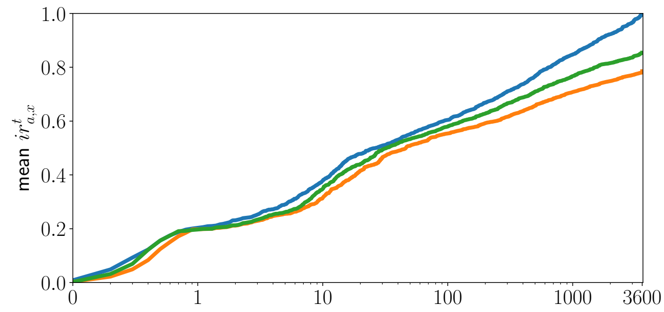
Figure 7.2: Comparison of different learners

which achieve the best results. The best approach is *regr.RRF* (an approach based on *Random Forest*). It finds the best algorithm on 230.5 instances in average (which represents 42.0% of instances).

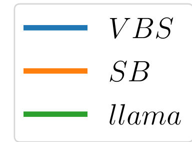
Table 7.2b shows the mean of the misclassification penalties over all the instances for measure m_t (for the same learners as in table 7.2a). We also see that there are differences between learners. However, the results are less interpretable: the value of the $m_t(i, VBS(i))$ for a given instance i corresponds to the minimum time needed by an algorithm to find the reference solution. Hence the misclassification penalty of a solver s on an instance i corresponds to the time needed by the algorithm chosen by s to find the reference solution minus the minimum time needed by any algorithm to find the reference solution if s finds the reference solution on i. If s does not find the reference solution, then the misclassification penalty corresponds to $3600/\min(ir_{a_i}^{3600}, 0.1)$ minus the minimum time needed by an algorithm to find the reference solution. We display here the mean misclassification penalty instead of the sum in the previous case to have something more interpretable. This time, *regr.rsm* (an approach based on *Response Surface Regression*) achieves the best results. Hence we keep this learner for the next experiments.

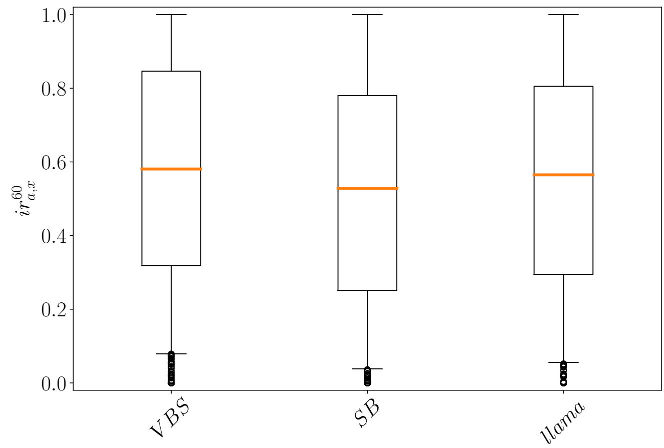
7.5 Llama results

This section uses per-instance algorithm selection for the problem $Q|r_j, brkdwn, s_{jk}, GC_{tight}| \sum T_g$. The algorithms which can be chosen are those used in the previous chapter (see section 6.2.10).

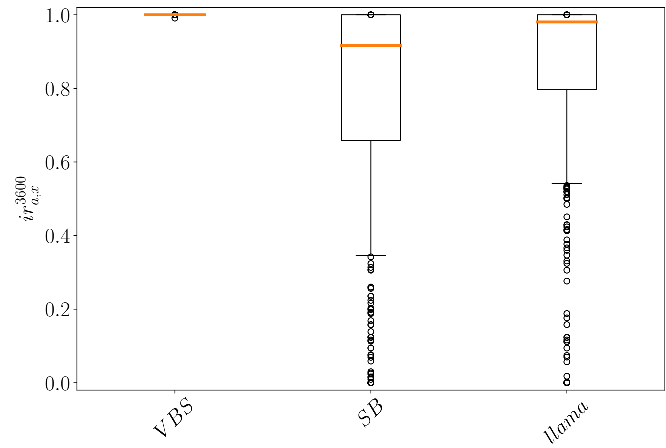


(a) Evolution of average inverse ratio

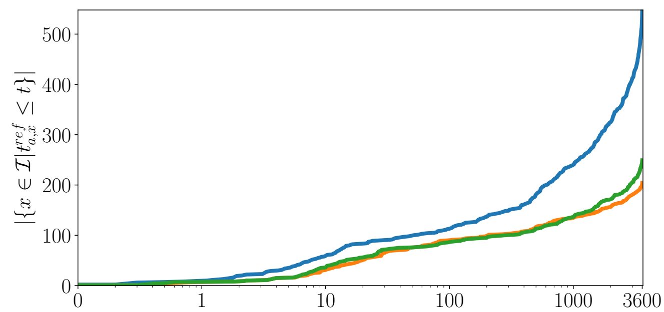




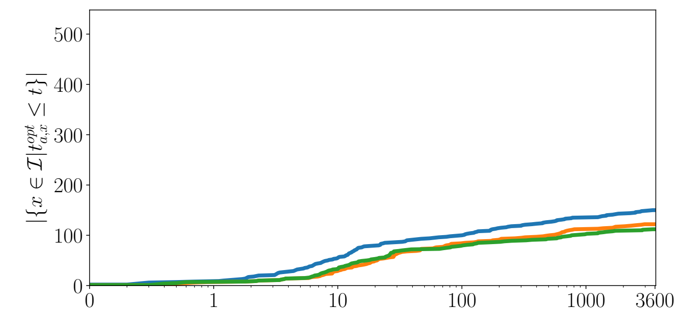
(b) Inverse ratios distribution after one minute.



(c) Inverse ratios distribution after one hour.



(d) Time to reach reference solution



(e) Time to prove solutions optimality

Figure 7.3: Results for per-instance algorithm selection

Algortihm	VBS	$llama$	SB
ACO_0	17	3	0
ACO_J	108	119	0
ACO_P	98	182	0
ACO_T	50	42	0
$ACO-Tabu_J$	109	169	548
$ACO-Tabu_T$	103	16	0
$CPO-ACO$	14	16	0
$Tabu_I$	4	0	0
$Tabu_S$	45	1	0

Figure 7.4: Algorithms selected by each selection method

We compare three methods :

- The *virtual best solver*, denoted VBS
- The *single best solver*, denoted SB. The *single best solver* is the algorithm (among those that can be selected) that minimizes the considered measure over all the instances, *i.e.*, $SB = \underset{a\in\mathcal{A}}{argmin} \sum_{i\in\mathcal{I}} m_t(i,a)$. In our case the single best solver is the method $ACO-Tabu_J$ (see results in sec. 6.2.10)
- A per-instance algorithm selector solver denoted *llama*

We use 10-fold cross-validation to evaluate the performance of *llama* (*i.e.*, our benchmark is randomly partitioned into 10 subsets, and we repeat 10 experiments where 9 subsets are used to train a selection model which is evaluated on the remaining subset).

The results are shown in Fig. 7.3.

llama outperforms SB. In particular the mean inverse ratio for llama after one hour is equal to $ir^{3600}_{llama} = 0.86$ whereas this ratio for the single best is equal to $ir^{3600}_{SB} = 0.79$ (which represents an increase of $(0.85 - 0.79)/0.79 = 7.5\%$). Similarly, the reference solution is found on 249 instances when using *llama* whereas SB finds it on 204 instances (which represents an increase of 22%). It is also worth mentioning that *llama* has an average misclassification penalty equal to 2913 whereas SB average misclassification penalty is equal to 4224 (which represents an increase of 45%).
We also observe that there is still room between *llama* and VBS for improving the model used to select configurations (VBS finds the reference solution on the 548 instances, its mean inverse ratio after one hour is equal to 1 and its average misclassification penalty is equal to 0 by definition). Maybe results could be improved by adding more features to describe instances. Another possibility could be to use other machine learning algorithms to learn the model.

Fig 7.4 shows the number of instances on which each algorithm has been selected by each selector. We especially notice that *llama* seems to select too often the ACO_P method in comparison with VBS.

7.6 Discussion

In this chapter we evaluate the interest of using *automatic algorithm selection* in order to automatically select the best algorithm for each instance. In particular we introduce some features for our instances, and especially for constraints such as sequence-dependent setup-times, scheduled breakdowns and GC constraints. Such features can be reused for any algorithm considering parallel machine scheduling problems. We also compare two performance measures in order to find

compromise between quality of solution after one hour of computation and speed at which good solutions are found. Our experimental results show that using *automatic algorithm selection* on a particular problem allows us to improve results, compared to the single best algorithm, though we are still rather far from the virtual best solver.

In all the problems considered so far, we assume that all jobs are known beforehand. However in our industrial context this is often not the case. Hence, in the next chapter we study the $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$ in a dynamic environment.

Chapter 8

Experimental Evaluation for Dynamic Scheduling

Contents

8.1 Motivation and definition of the problem 133
8.2 Review of dynamic scheduling problems 133
8.3 Objectives and framework 135
8.4 Experimental results 138
8.5 Towards anticipation 140
8.6 Discussion 141

In previous chapters, we considered that all the problems data were known beforehand. However, in practice this is often not the case. Hence algorithms should react to the arrival of new jobs, or to the fact that some jobs are realized on the shop floor. This chapter aims at studying one particular scheduling problem in a dynamic environment.
In section 8.1 we give further motivation to consider the dynamic problem. In section 8.2 we give a short survey concerning dynamic optimization. In section 8.3 we describe our dynamic framework and the studied objectives. In section 8.4 we show our experimental results. Finally in section 8.5 we evaluate the gap between static and dynamic approaches and give some insights about methods which could help reduce this gap.

8.1 Motivation and definition of the problem

As aforementioned, the scheduling problems dealt with in this thesis concern order preparation. In order preparation, orders come over time. In particular, some orders arrive during the day and must be shipped before the end of the same day. In the previous chapters, we considered offline problems where all jobs were known, and we used release dates to ensure that a job does not start before its release date. However, in our industrial context, the release date r_j of a job j actually corresponds to the time where j is revealed: at time $t < r_j$, we do not know that j will be revealed. Hence, we cannot schedule all jobs at once: before the beginning of the day, at time t_0, we can only schedule every job j such that $r_j \leq t_0$; then each time some new jobs are revealed, we can schedule these new jobs. Fig 8.1 shows the distribution of the release dates, i.e., the number of jobs that are revealed during each half-hour in the day (accumulated over all the instances). In particular, it is noticeable that only 51% (in average) of the jobs are known at the beginning of the day.

8.2 Review of dynamic scheduling problems

Scheduling problems considering uncertainty have been widely studied. Consequently, many terms are used to describe the different problems, and depending on the authors, sometimes different terms are used to describe the same entity, or

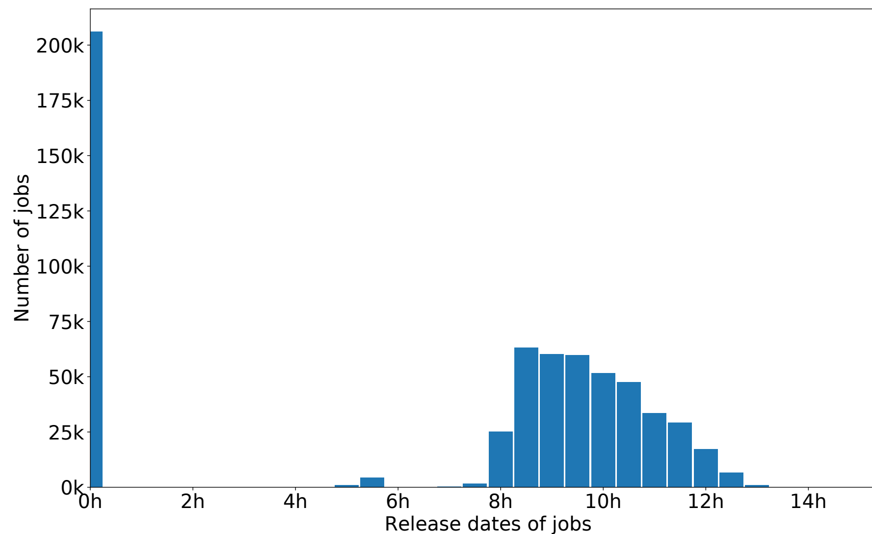


Figure 8.1: Number of jobs which arrived at each time of the day (grouped in interval of 30 minutes) accumulated over the 548 instances.

sometimes a similar term is used to describe different entities.
We can first mention the online scheduling problems [Liu09, PST04, LXCZ09]. In such a context, one tries to find algorithms whose performance is bounded with respect to offline algorithm performance. In particular, this research field is not interested in studying a given algorithm's performance over a given data-set, but it aims to calculate bounds for algorithms that are valid for every instance of a given scheduling problem.

Another approach to scheduling problems under uncertainty is the dynamic approach. [HL05] gives a survey about project scheduling under uncertainty. According to the scheduling environment, the authors describe two types of methods for dynamic problems: proactive (robust) methods and reactive methods. In an environment where jobs' duration is unknown (but jobs themselves are known), proactive (robust) methods are the most studied. These methods establish schedules as robust as possible before any uncertainty is revealed. It means that one establishes a schedule, and we have some probabilistic guarantees that it will not change once the uncertainty will be revealed. The objective is often to optimize the mean of a function over the different possible scenarios.

By contrast with *proactive methods*, there are the *reactive methods*, in which a first schedule is generated, and then it is modified each time a new job comes. Depending on the rescheduling time, methods are split into two categories. On the first hand, we have the so-called *predictive-reactive* scheduling, which consists in calculating an initial schedule and applying next a repair function each time a new job comes. The repair function must be a polynomial function (polynomial according to the size of the instance); in particular, the time needed for its application should be negligible. Most of the time, the repair function only inserts the new jobs in the current schedule. For example the method for inserting new activities given by [AR00] falls into this category.

On the other hand, we have *full rescheduling* methods, consisting of recalculating a complete solution each time an uncertain event occurs. An interesting survey about dynamic scheduling in manufacturing systems can be found in [OP08]. Furthermore, [BVLB09] gives a theoretical framework for such scheduling, without experimental evaluation nonetheless. [DDH11] gives a tree-based exact search technique in order to deal with such problems.

[GMW16] gives a survey of rescheduling strategies (with a specific focus on articles that do not deal with full-rescheduling). Both [GMW16] and [OP08] emphasize the question of 'when to reschedule'. This question arises in many articles, with varying conclusions. [PF17, SB99] show that rescheduling as frequently as possible leads to better results, whereas

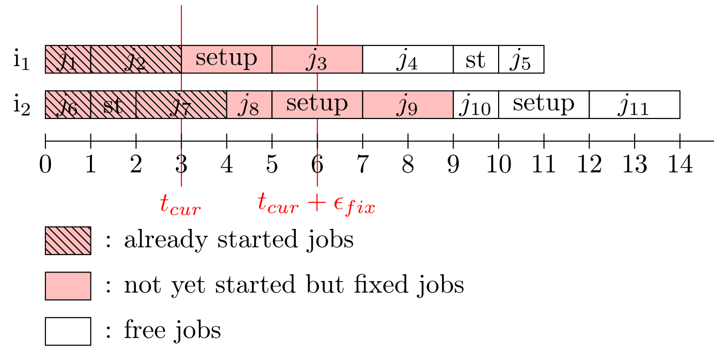


Figure 8.2: Example of a schedule with the already executed part, the fixed part and the free part.

[GM16, MK94] show that scheduling too frequently and not frequently enough decreases performance. More recently [SFRS20] shows that a rescheduling policy's performance depends on the plant characteristics, and therefore care should be taken when selecting a rescheduling policy.

With the question of 'when to reschedule' comes the question of 'how long lasts the rescheduling'. This question has received little attention in the articles mentioned above. Depending on the application, the question is not necessarily relevant. If the jobs' duration is far greater than the time needed for the computation of solutions, then the question has little interest. For example, in [SFRS20], schedules are generated for several months, the magnitude of the jobs' duration is hours, and the computation of solutions is limited to 15 minutes a day. In such a context, the events which happen during the computation of solutions are negligible. In our case, some jobs last few seconds; hence even if we use an algorithm for several minutes, the context changes during the algorithm's execution. This question has been addressed in the Vehicle Routing Problem (VRP) community. [PGGM13] gives a review on the Dynamic Vehicle Routing Problem. In particular, it distinguishes between periodic re-optimization and continuous re-optimization. In the latter case, optimization algorithms are running continuously, and new information (new customer arrival or end of the current customer service) is integrated continuously. Algorithms for continuous re-optimization are also described in [PGM12b, PGM12a, BVH04b].

Another question often addressed is the question of the nervousness when the schedule changes. Articles [KCEP08, NH10] deal with this problem. They include in their objective function the cost of perturbing the current schedule. Hence the instability of the schedule (and the nervousness of workers associated with it) is considered.

8.3 Objectives and framework

In this chapter we consider only the $Q|r_j, s_{jk}, GC_{tight}|\sum T_g$ problem in a dynamic context. The dynamics of the problem come from the fact that jobs are revealed during the algorithm's execution. As aforementioned, several strategies exist to select 'when to reschedule'. Here we only consider the 'event-based' rescheduling; it means that we allow a rescheduling each time a new job is revealed. 'Periodic rescheduling', as mentioned above, is relevant when we consider a rolling horizon, but in our case, the entire horizon (which corresponds to a full day of work) is always considered.

In this chapter, we evaluate two objectives. First, as mentioned above, and as seen with Infologic's clients, the schedule's stability is essential. From a practical point of view, operators are aware of their tasks' near future. It means that they know the tasks they must perform in the next following minutes, but they are not interested in the tasks which must start in several hours. Hence, whenever the schedule changes, the tasks that start in the next minutes must not change. Hence a first goal is to evaluate the loss of forbidding the jobs which must start in the ϵ_{fix} next seconds from moving, for different values of ϵ_{fix}.
Our second goal is to evaluate the impact of the time let to algorithms to compute new schedules. Hence each time a new

job is revealed, we let δ_{dur} seconds to the algorithm to compute new schedules. Our goal is to evaluate the influence of this δ_{dur} parameter. It is crucial to notice that δ_{dur} can be far greater than the jobs' duration. Hence while the algorithm is computing new schedules, some jobs are started by the operators. In this case, the algorithm cannot produce new schedules in which the start time of those jobs changes.

In our framework, we always have a *current schedule*, denoted s_{cur}, which is the schedule followed by the workers on the shop floor. During the simulation, the current time is denoted t_{cur}. There are two types of event which modify the data of the problem solved by our algorithm:

(1) A new job is revealed;

(2) A job is fixed because its start time (more precisely the start of its setup-time) becomes lower than $t_{cur} + \epsilon_{fix}$;

These two events are consequences of different actors. (1) is linked with customers' demands whereas (2) is the consequence of the time clock. As soon as they happen, events are handle by the algorithm.

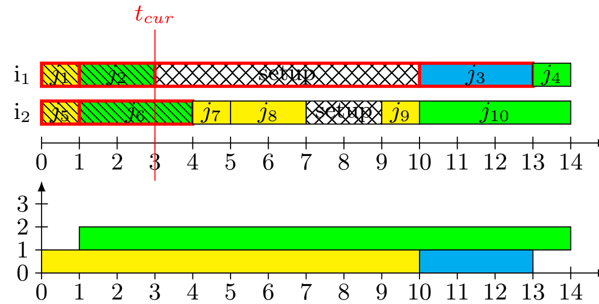
Anytime algorithm: The algorithm triggered after each event must be anytime, *i.e.*, it improves the incumbent solution in a continuous fashion until its execution is halted by a new event (or because it has proven that the incumbent solution is optimal). Any algorithm introduced in chapter 2 can be used. However, in this chapter we focus on the $CPO - ACO$ algorithm as results described in section 6.2.8 show that this algorithm achieves good results on the considered problem.

Handling of event (1): Whenever a new job j is revealed, the anytime algorithm is stopped. The new job is inserted in the current schedule using a greedy insertion heuristic. It is important to ensure that the newly inserted job is such that $B_j^{st} \geq t_{cur} + \epsilon_{fix}$, where B_j^{st} is the start of the setup-time of j. In particular, if j is inserted on a machine i such that the last job of i ends at t_l with $t_l < t_{cur} + \epsilon_{fix}$, then an idle time of duration $t_{cur} + \epsilon_{fix} - t_l$ must be introduced on machine i between the end of its last job and the start of j. Once the new job's insertion is ended, the anytime algorithm can resume and produce new schedules (which contain job j).

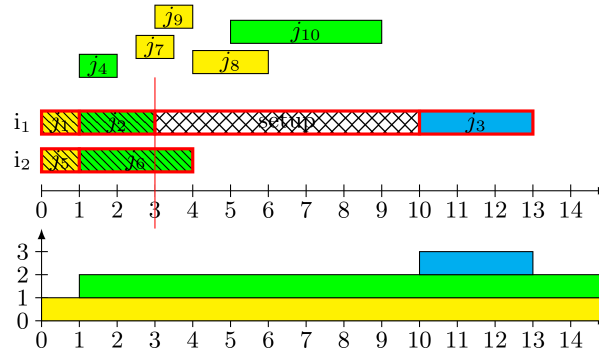
Handling of event (2): Whenever the start of the setup-time of a job j, $B_j^{st,cur}$, in the current schedule is such that $B_j^{st,cur} \leq t_{cur} + \epsilon_{fix}$, the job j becomes fixed. It is removed from the free jobs list. Hence in every new schedule, the value of the start and end times of j and its machine cannot changed.
The anytime algorithm continuously improves the free jobs' start and end times to minimize the sum of jobs' tardiness. It ends at time $h - \epsilon_{fix}$ where h is the end of the day (the horizon).
Fig 8.2 shows an example of a current schedule, with the fixed variables and the free variables.

Insertion heuristic: The insertion heuristic used is the $ATCS$ described in section 2.1. Hence, in the simplest case, when a single job is revealed, it is scheduled at the end of the machine that ends the soonest. However, in most cases in our data-set, jobs come in a batch (all jobs of one group are revealed together). In that case, we select the machine that ends the soonest and the job that best fits this machine (according to the $ATCS$ formula). We repeat the process until all jobs are inserted.

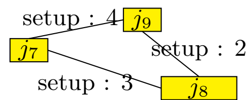
Fixed jobs: The anytime algorithm knows the jobs which are fixed. Hence, this information must be exploited to generate schedules that respect the fixed jobs.
For the CPO solver, each time a job j is fixed (with values B_j, C_j and I_j for its start time, end time, and machine), we stop the solver, and we reduce the domain of the variable a_j (which represents the job j see section 2.5.5 for more information) so that it contains the only value B_j for the start time and only the value C_j for the end time, and we make the interval $a_j^{I_j}$ present (to impose the machine of j).
There exist works that aim at removing the realized variables in order to speed up the search [EGS+20]. In our case, we could have used such techniques by removing each job as soon as it is fixed and only memorizing the group it belongs to.



(a) At time $t_{cur} = 3$ with $\epsilon_{fix} = 0$, jobs j_1, j_2, j_5 and j_6 are fixed and the setup-time of job j_3 starts, hence we also fix job j_3 (fixed jobs have red borders).

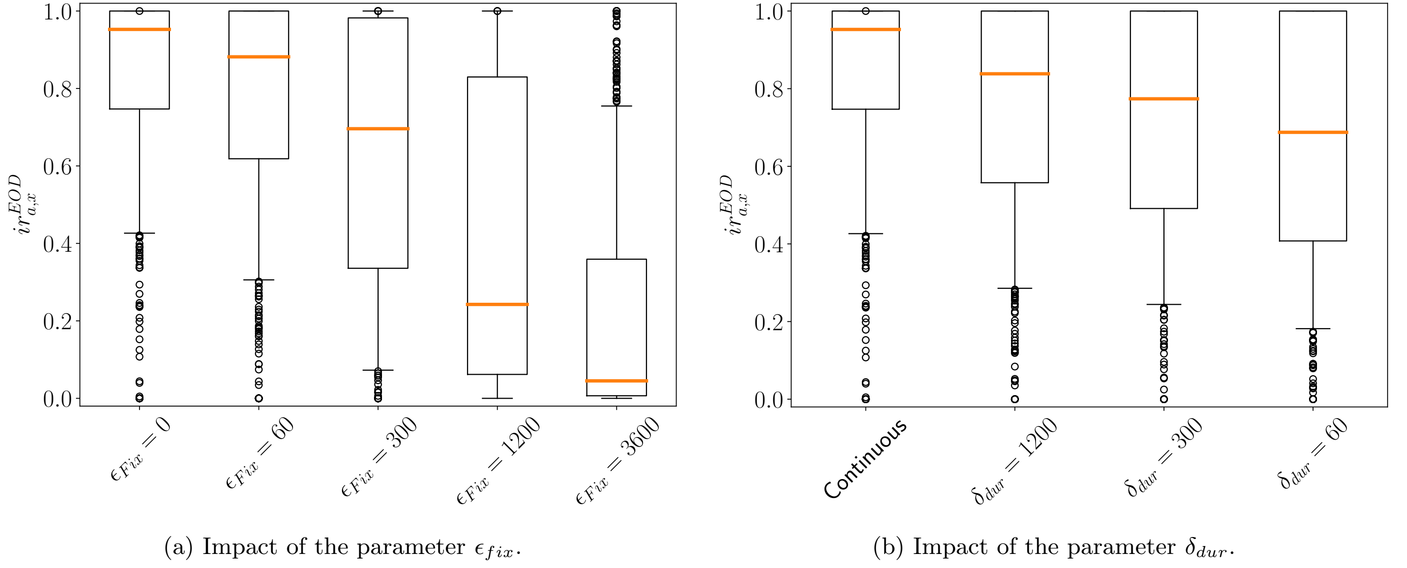


(b) When we try to build new schedules (i.e., by re-ordering the free jobs), we remark that neither the green group nor the yellow one is ended at time $t = 10$ when job j_3 starts. Hence the three groups can potentially be active at time $t = 10$. Hence whenever the *anytime algorithm* builds a new schedule, it must ensure that either the green group or the yellow one ends before $t = 10$.



(c) Ensuring that either the green group or the yellow one ends before $t = 10$ is not easy. Indeed, let us assume that one wants to end the yellow group before $t = 10$. The three jobs j_7, j_8 and j_9 must be executed between time $t = 4$ and $t = 10$. Because of sequence-dependent setup-times between those jobs, to make the yellow group end before $t = 10$, one must find a permutation of the unscheduled yellow jobs such that the sum of their durations plus the sum of their setup-times is lower or equal to 6. It is equivalent to solving the decision version of the problem $1|s_{ij}|C_{max}$, which is known to be $\mathcal{NP}$-Complete [Pin16]

Figure 8.3: A schedule with a fixed part, setup-times and a *GC* constraint. The *GC* constraint ensures that at most two groups are active simultaneously.



(a) Impact of the parameter ϵ_{fix}.

(b) Impact of the parameter δ_{dur}.

Figure 8.4: Impact of the two parameters ϵ_{fix} and δ_{dur}. Distribution of the inverse ratios $ir_{a,x}^{EOD}$ for the different considered approaches.

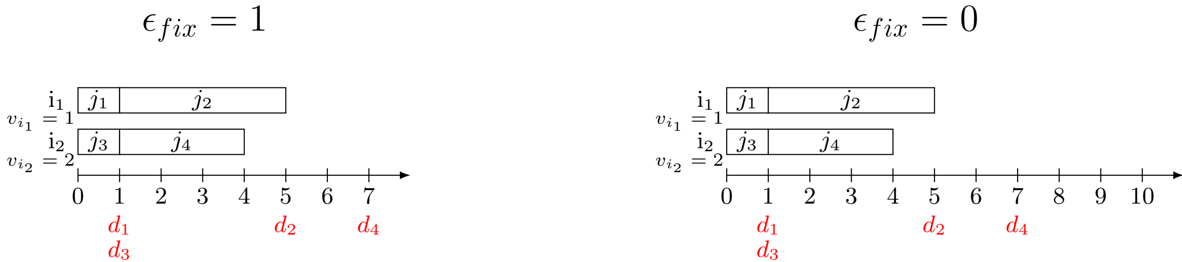
Also, whenever the last job of a group is fixed, we can remove all the information concerning that group. However, for the sake of simplicity, we do not consider such techniques in this work.

Special case when considering setup-times and GC: As aforementioned, each time a new job j verifies $B_j^{st,cur} \leq t_{cur} + \epsilon_{fix}$, j becomes fix. When combining setup-times and GC, the problem is more complicated. It is possible that when we fix some jobs because their setup-times start, then finding a solution which respects those fixed jobs becomes $\mathcal{NP} - complete$ problem. Such an example is illustrated in Fig 8.3.
Hence, to avoid this problem in our online execution, we impose that whenever we fix a job, we also fix all the jobs that end during the execution of its setup-time. (In the example given in Fig 8.3 it means that when we fix job j_3 we also fix jobs j_7, j_8 and j_9).

8.4 Experimental results

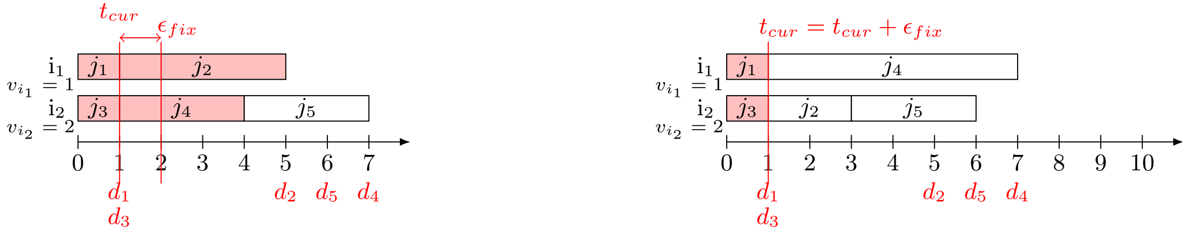
Performance measure: During one simulation, several problems are generated (one problem each time a new job is revealed and each time a job is fixed). However we are only interested in the last generated schedule (and not in the intermediate ones). Indeed, in the applicative context we are only interested in knowing, at the end of the day, the total lateness over all the jobs for the realized schedule (and not for the intermediate ones). Let x be an instance and a an algorithm, we denote x_a^{EOD} (for **E**nd **O**f **D**ay) the value of the sum of tardiness over all the jobs in the last realized schedule. Similarly to what we do for the static case, we denote the reference solution $x^{EOD,*}$ the best solution found by any of the considered algorithm ($x^{EOD,*} = \underset{a}{min}\ x_a^{EOD}$). We also compute the inverse ratio at time EOD, and we denote it $ir_{a,x}^{EOD}$.

Results analysis: In Fig. 8.4a we compare the results for different values for ϵ_{fix}. For this experiment, the anytime algorithm is run continuously. It means that it is never stopped between two arrivals of jobs. As expected, we see that the greatest the value of ϵ_{fix} the worst the results (when ϵ_{fix} grows, the problems are more constrained, which generally leads to worse solutions). However, we can observe that the gap between approaches grows dramatically as ϵ_{fix} grows.



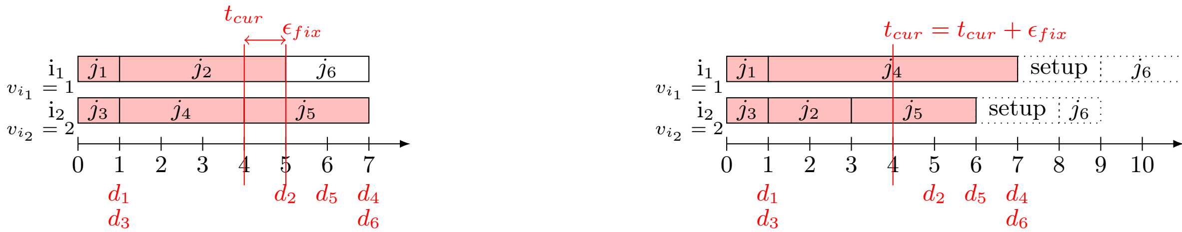
(a) At the first step, an initial schedule is computed whose total tardiness is equal to zero. Hence this schedule is optimal.

(b) The initial schedule is the same for both cases, as the problems are identical.



(c) A second schedule is generated at $t = 1$ when job j_5 is revealed. At that time, with $\epsilon_{fix} = 1$, the pink jobs are fixed. Hence j_5 is scheduled on i_2 and has one unit of tardiness. This schedule is optimal considering the fixed jobs.

(d) At $t_{cur} = 1$ with $\epsilon_{fix} = 0$ jobs j_2 and j_4 are not fixed. Hence it is possible to generate a schedule whose total tardiness is zero (jobs j_4 and j_2 are swapped and job j_5 is scheduled at the end of i_2)



(e) At $t_{cur} = 4$ j_6 is revealed and all the other jobs are fixed. j_6 is scheduled on i_1 and completed in time. So the total tardiness of this schedule is equal to 1 (only j_5 is late).

(f) At $t_{cur} = 4$ with $\epsilon_{fix} = 0$ all the jobs are also fixed. However, whichever the machine for j_6, a two units setup-time is needed (because in that case j_6 is either after j_4 or j_5). In either case, j_6 has a lateness greater than 1. So the schedule generated with $\epsilon_{fix} = 0$ is worse than the one generated with $\epsilon_{fix} = 1$, even if all the intermediate generated schedules are optimal.

Figure 8.5: An example where a larger value for ϵ_{fix} leads to a better final schedule. The six jobs are j_1 (resp. j_2, j_3, j_4, j_5 and j_6) whose processing time is 1 (resp. 4, 2, 6, 6 and 2), whose due-date is 1 (resp. 5, 1, 7, 6, 7) and release date is 0 (resp. 0, 0, 0, 1, 4). The machines are i_1 (resp; i_2) whose speed is 1 (resp. 2). On the left column $\epsilon_{fix} = 1$ whereas on the right one $\epsilon_{fix} = 0$. The setup-times are all equal to zero except between j_4 and j_6 and between j_5 and j_6, where it is equal to 2.

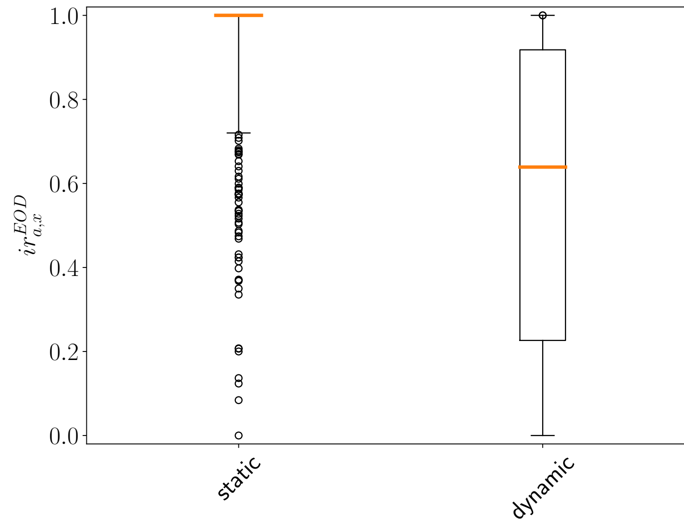


Figure 8.6: Comparison of static and dynamic cases

When the next hour cannot be changed ($\epsilon_{fix} = 3600$), three quarters of the instances have an inverse ratio lower than 0.36. With lower values for ϵ_{fix}, the gap is more acceptable. For example, if only the next 5 minutes cannot be changed, half of the instances have an inverse ratio greater than 0.70. Hence this shows that if a large part of the schedule is fixed, then the quality of the schedule decreases dramatically. It should be put in comparison with the increase in nervousness for the operators when changes in near futures occur (which is difficult to quantify)
It is also noticeable that, when $\epsilon_{fix} = 0$, half of the instances have an inverse ratio lower than 0.95. One could expect that with the lowest ϵ_{fix} the approach would always find the best solution. It is important to notice that, in some cases, constraining the sub-problems can lead to better results, even if the algorithm is able to find an optimal solution for every sub-problem. An example of such a case is depicted in Fig. 8.5. This can explain why the approach with $\epsilon_{fix} = 0$ does not always find the reference solution.

In Fig. 8.4b, we compare the results for different values of δ_{dur}. As a reminder, each time a new job is revealed, we start the anytime algorithm, and we let it run δ_{dur} seconds to find new (improving) schedules. For these simulation, ϵ_{fix} is set to 0. Once again, as expected, we observe that the longer the processing, the better the results. However, the gap between the approaches is less important than in the previous case. For example, when letting only 60 seconds after each new job arrival, half of the instances have an inverse ratio greater than 0.69. Once again, the continuous approach does not always find the reference solution. It could also be explained by the fact that finding intermediate sub-optimal solutions sometimes lead to better results than always finding the optimal solution for each sub-problem.

8.5 Towards anticipation

Fig 8.6 compares the results for the dynamic case (with a continuous run and $\epsilon_{fix} = 0$) and the static case for which all jobs are known at the beginning of the day. We observe that the gap is important between the two cases, letting room to improve the dynamic version of $CPO - ACO$. It is noticeable that for 114 instances (over the 548), the static algorithm finds solutions worse than the dynamic one. Such a result can be explained by the time let to each algorithm. In the static case, the algorithm is run for one hour. In the dynamic case, the algorithm is run several times on sub-problems, but when considering the total time it can represent several hours.

Fig 8.7 shows the correlation between our instances. To build it we define a distance measure between our instances. To do so, we first define a distance measure between two jobs (the exact definition of this distance is out of the scope of this

thesis) and then, given two instances, we look for a perfect matching with minimal weight between the two sets of jobs (adding some dummy jobs to the instance with the lowest number of jobs).

We observe some similarities between our instances, with subgroups among the instances. In particular, we observe similarities between the instances which concern the same day of a week (Mondays are similar to Mondays, Tuesdays are similar to Tuesdays, ...). We furthermore observe that Tuesdays and Thursdays also have similarities.

Exploiting these correlations by anticipating the jobs to come could help reduce the gap between dynamic and static algorithms. Stochastic methods [HB06, BVH05, BVH04a] could be used to do so. In particular these methods compute some statistics on the past day of work in order to anticipate the more probable scenarios.

8.6 Discussion

In this chapter, we first show the need of studying problems in a dynamic environment (because of the jobs' arrival). We also give a survey about dynamic approaches. Here, we are especially interested in evaluating the impact of preventing the close future from changing. We remark that the longest the fixed part the worst the results. In particular, the decrease in solution quality is really important when a long part of the schedule cannot move. We also evaluate the impact of the computation duration. Finally we observe that the gap between static and dynamic results is considerable. To fill this gap, we decide to measure the distance between our instances. We remark that there exists some regularities in our data. Hence this can motivate the use of stochastic methods in order to solve our instances.

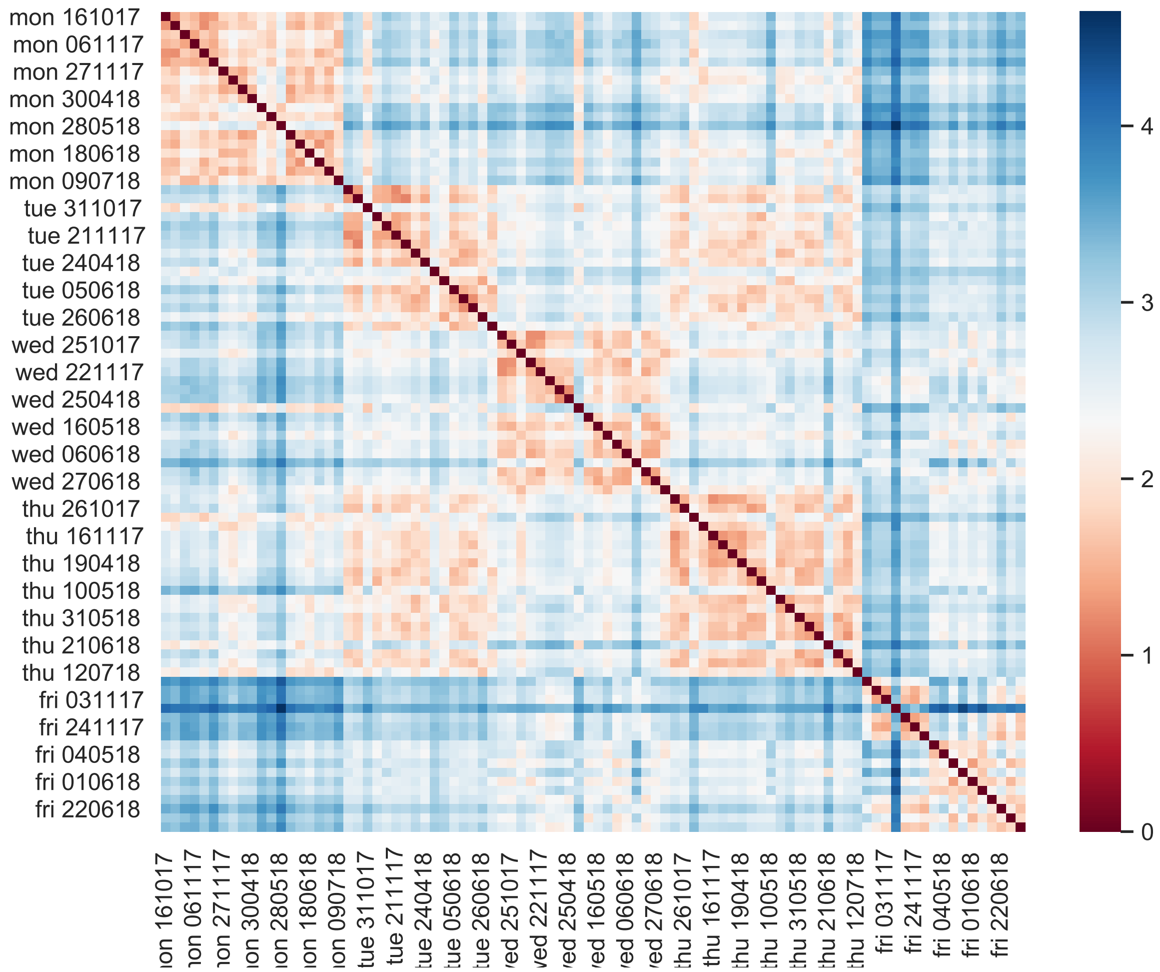


Figure 8.7: Distance between each instance.

Chapter 9

Conclusion

In this thesis, we aimed at solving real uniform parallel machine scheduling problems related to order preparation in agrifood warehouses. Our solver is integrated within an ERP (*Copilote*, developed by *Infologic*) which is used in many different kinds of warehouses. Hence, it must be able to handle different kinds of constraints and objective functions. In particular, we have introduced the *Group Cumulative* constraint, which may be viewed as a cumulative constraint on groups of jobs. We have studied the theoretical complexity of problems related to this constraint, and we have introduced new algorithms for handling this constraint. These new algorithms are based on three main kinds of approaches (Ant Colony Optimization, Tabu Search, and Constraint Programming), and hybrid approaches where ACO is combined with Tabu Search or CP.

We have introduced a new benchmark, based on real data, for ten different scheduling problems, ranging from a simple assignment problem, to more constrained problems with release and due dates, different machine speeds, sequence-dependent setup times, breaks, and/or group cumulative constraints. We have experimentally evaluated and compared twelve different algorithms on these ten problems, in order to study strengths and weaknesses of different solving approaches on various kinds of constraints. In particular, we have shown that exact approaches are very efficient for some problems (*e.g.*, Integer Programming for the assignment problem, or CP for scheduling problems with release and due dates), whereas hybrid approaches are better suited for solving problems with more constraints (*e.g.,* CPO-ACO for problems with sequence dependent setup times, or ACO-Tabu for problems with both sequence dependent setup times and breaks).

As our goal was to design a method which can be applied to solve all the scheduling problems faced by *Copilote*, and because the best algorithm to use depends on the considered scheduling problem, we have evaluated the interest of using an automatic algorithm selection approach called *Llama.* As a proof of concept on one of our ten problems, we have shown that using *Llama* can lead to substantial benefits.

Finally, we also studied the applicability of our hybrid *CPO-ACO* to a dynamic scheduling problem where jobs are revealed during the day. In particular, we have evaluated the impact of freezing jobs that have short term start dates (in order to avoid stressing workers) on the objective function.

Perspectives

Comparing new algorithms: We compare essentially four algorithms (or family of algorithms): *ACO*, *Local Search* (and especially *Tabu Search*), *Linear Programming* and *Constraint Programming*, plus two hybrids methods: *CPO-ACO* and *ACO-Tabu.* It could be interesting to consider other approaches, especially *genetic algorithm* which are often used in scheduling problems. We also observe that *Linear Programming* has bad results on our data-set, it could be interesting to consider *column generation* which is also often used to solve scheduling problems. Another idea would consist in evaluating the multi-threaded version of algorithms.

Using *GC* with varying consumption: In the *GCSP* presented in this thesis, each group consumes one unit of the cumulative resource. It could be interesting to study the case where each group can consume a different number of units of the cumulative resource. Such a case should be interesting from an industrial point of view: here we consider that only one pallet is used to send all the products of an order to the client. However, in some cases, several pallets are used for a unique order.

Implementing a dedicated propagator: In order to consider the *GC* constraint in *CPO* we use a decomposition of the constraint using the existing *span* and *cumulative* constraints. Another approach could consist in implementing a propagator dedicated to the *GC* constraint.

Considering other constraints: We consider some constraints in this work. However there exist many other constraints that we do not consider. For example we could consider the case where the speeds of machines vary with time. One perspective could be to consider those constraints. Another perspective could be considering multi-stage problems and evaluating the methods presented here for these problems.

Considering other instance features: We consider some instance features in order to use *automatic algorithm selection*. However we observe that there is room between the *virtual best solver* and *llama*. Maybe, by adding other features to our instances we could improve the performance of *automatic algorithm selection*.

Extending *automatic algorithm selection* to all the considered scheduling problems: We study the interest of *automatic algorithm selection* only on problem $Q|r_j, brkdwn, s_{jk}, GC_{tight}| \sum T_g$. It could be relevant to study it on our whole data-set with the ten problems considered in chapter 6. In particular it would be interesting to see if learning on a problem could help choosing efficient algorithms (or efficient parameters) for another problem.

Evaluating stochastic methods on our data-set: As stated in chapter 8, the gap between static and dynamic algorithms is considerable. Moreover there exist regularities in our data-set and hence a part of the jobs to come could be predictable. Hence, it could be interesting to evaluate stochastic methods on our benchmark.

Bibliography

[AA14] Muminu O. Adamu and Aderemi O. Adewumi. A survey of single machine scheduling to minimize weighted number of tardy jobs. *Journal of Industrial & Management Optimization*, 10(1):219, 2014.

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, April 1993.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge ; New York, 1st edition edition, April 2009.

[ADN08] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE/Wiley, 2008.

[All15] Ali Allahverdi. The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, 246(2):345–378, October 2015.

[AR00] Christian Artigues and François Roubellat. A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. *European Journal of Operational Research*, 127(2):297–316, December 2000.

[Art12] Christian Artigues. Scheduling and (Integer) Linear Programming. *CPAIOR 2012 international master class*, page 168, 2012.

[AS09] Fernando Masanori Ashikaga and Nei Yoshihiro Soma. A heuristic for the minimization of open stacks problem. *Pesquisa Operacional*, 29(2):439–450, August 2009.

[AST09] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Principles and Practice of Constraint Programming - CP 2009*, Lecture Notes in Computer Science, pages 142–157, Berlin, Heidelberg, 2009. Springer.

[BB18] Philippe Baptiste and Nicolas Bonifas. Redundant cumulative constraints to compute preemptive bounds. *Discrete Applied Mathematics*, 234:168–177, January 2018.

[BDM+99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, January 1999.

[BKK+16] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, August 2016.

[BLK+16] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. Mlr: Machine learning in R. *The Journal of Machine Learning Research*, 17(1):5938–5942, January 2016.

[BM07] Stephen Boyd and Jacob Mattingley. Branch and bound methods. *Notes for EE364b, Stanford University*, pages 2006–07, 2007.

[Bon17] Nicolas Bonifas. *Geometric and Dual Approaches to Cumulative Scheduling, Optimization and Control [Math.OC]*. PhD thesis, Université Paris-Saclay, 2017.

[BPRR11] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, September 2011.

[BT94] Roberto Battiti and Giampietro Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2):126–140, May 1994.

[BVH04a] Russell Bent and Pascal Van Hentenryck. The value of consensus in online stochastic scheduling. In *ICAPS*, volume 4, pages 219–226, 2004.

[BVH04b] Russell W. Bent and Pascal Van Hentenryck. Scenario-Based Planning for Partially Dynamic Vehicle Routing with Stochastic Customers. *Operations Research*, 52(6):977–987, December 2004.

[BVH05] Russell Bent and Pascal Van Hentenryck. Online stochastic optimization without distributions. In *ICAPS*, volume 5, pages 171–180, 2005.

[BVLB09] Julien Bidot, Thierry Vidal, Philippe Laborie, and J. Christopher Beck. A theoretic and practical framework for scheduling in a stochastic environment. *Journal of Scheduling*, 12(3):315–344, June 2009.

[BYBS10] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-Race and Iterated F-Race: An Overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, Berlin, Heidelberg, 2010.

[BZ09] J. H. Bartels and J. Zimmermann. Scheduling tests in automotive R&D projects. *European Journal of Operational Research*, 193(3):805–819, March 2009.

[CDM91] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed Optimization by Ant Colonies. In *Proceedings of the First European Conference on Artificial Life*, January 1991.

[CFBA16] Carlos J. Costa, Edgar Ferreira, Fernando Bento, and Manuela Aparicio. Enterprise resource planning adoption and satisfaction determinants. *Computers in Human Behavior*, 63:659–671, October 2016.

[CGT95] Runwei Cheng, Mitsuo Gen, and Tatsumi Tozawa. Minmax earliness/tardiness scheduling in identical parallel machine system using genetic algorithms. *Computers & Industrial Engineering*, 29(1):513–517, September 1995.

[Cha18] Maxime Chabert. *Constraint Programming Models for Conceptual Clustering : Application to an Erp Configuration Problem*. These de doctorat, Lyon, December 2018.

[CHG19] Pedro M. Castro, Iiro Harjunkoski, and Ignacio E. Grossmann. Discrete and continuous-time formulations for dealing with break periods: Preemptive and non-preemptive scheduling. *European Journal of Operational Research*, 278(2):563–577, October 2019.

[CK16] Imran Ali Chaudhry and Abid Ali Khan. A research survey: Review of flexible job shop scheduling techniques. *International Transactions in Operational Research*, 23(3):551–591, May 2016.

[Cla99] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, May 1971. Association for Computing Machinery.

[Cot06] Richard W. Cottle. George B. Dantzig: A legendary life in mathematical programming. *Mathematical Programming*, 105(1):1–8, January 2006.

[Dan90] George B. Dantzig. Origins of the simplex method. In *A History of Scientific Computing*, pages 141–151. Association for Computing Machinery, New York, NY, USA, June 1990.

[DDE+05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software. *Proceedings of the IEEE*, 93(2):293–312, February 2005.

[DDH11] Filip Deblaere, Erik Demeulemeester, and Willy Herroelen. Reactive scheduling in the multi-mode RCPSP. *Computers & Operations Research*, 38(1):63–74, January 2011.

[Des20] Cygnus Web Design. Problem instances. http://soa.iti.es/problem-instances, 2020.

[DG97] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[DGRU13] Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. A Hybrid ACO+CP for Balancing Bicycle Sharing Systems. In *Hybrid Metaheuristics*, Lecture Notes in Computer Science, pages 198–212, Berlin, Heidelberg, 2013. Springer.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. A Bradford Book. MIT Press, Cambridge, Mass., 2004.

[EGS+20] Alexander Ek, Maria Garcia de la Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack. Aggregation and Garbage Collection for Online Optimization. In *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 231–247, Cham, 2020. Springer International Publishing.

[EO11] Emrah B. Edis and Irem Ozkarahan. A combined integer constraint programming approach to a resource-constrained parallel machine scheduling problem with machine eligibility restrictions. *Engineering Optimization*, 43(2):135–157, February 2011.

[FL13] Kuei-Tang Fang and Bertrand M.T. Lin. Parallel-machine scheduling to minimize tardiness penalty and power cost. *Computers & Industrial Engineering*, 64(1):224–234, January 2013.

[FMM01] Paulo M França, Alexandre Mendes, and Pablo Moscato. A memetic algorithm for the total tardiness single machine scheduling problem. *European Journal of Operational Research*, 132(1):224–242, July 2001.

[FR95] Thomas A. Feo and Mauricio G. C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2):109–133, March 1995.

[FR12] Luis Fanjul-Peyro and Rubén Ruiz. Scheduling unrelated parallel machines with optional machines and jobs selection. *Computers & Operations Research*, 39(7):1745–1753, July 2012.

[GGJK78] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity Results for Bandwidth Minimization. *SIAM Journal on Applied Mathematics*, 34(3):477–495, 1978.

[GKL96] Fred Glover, James P. Kelly, and Manuel Laguna. New advances and applications of combining simulation and optimization. In *Proceedings of the 28th Conference on Winter Simulation - WSC '96*, pages 144–152, Coronado, California, United States, 1996. ACM Press.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. Elsevier, 1979.

[Glo95] Fred Glover. *Tabu Search Fundamentals and Uses*. Graduate School of Business, University of Colorado Boulder, 1995.

[GM02] Michael Guntsch and Martin Middendorf. A Population Based Approach for ACO. In *Applications of Evolutionary Computing*, volume 2279, pages 72–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[GM14] Roger J. Gagnon and Shona D. Morgan. Remanufacturing scheduling systems: An exploratory analysis comparing academic research and industry practice. *International Journal of Rapid Manufacturing*, 4(2-4):179–198, January 2014.

[GM16] Dhruv Gupta and Christos T. Maravelias. On deterministic online scheduling: Major considerations, paradoxes and remedies. *Computers & Chemical Engineering*, 94:312–330, November 2016.

[GMW16] Dhruv Gupta, Christos T. Maravelias, and John M. Wassick. From rescheduling to online scheduling. *Chemical Engineering Research and Design*, 116:83–97, December 2016.

[GNS20a] Lucas Groleaz, Samba N. Ndiaye, and Christine Solnon. Solving the Group Cumulative Scheduling Problem with CPO and ACO. In *Principles and Practice of Constraint Programming*, volume 12333, pages 620–636. Springer International Publishing, Cham, 2020.

[GNS20b] Lucas Groleaz, Samba Ndojh Ndiaye, and Christine Solnon. ACO with automatic parameter selection for a scheduling problem with a group cumulative constraint. In *GECCO 2020 - Genetic and Evolutionary Computation Conference*, pages 1–9, Cancun, Mexico, July 2020.

[GO21] LLC Gurobi Optimization. Gurobi optimizer reference manual. 2021.

[GPFW96] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey,. Technical report, CINCINNATI UNIV OH DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, January 1996.

[GPG02] C. Gagné, W L Price, and M. Gravel. Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times. *Journal of the Operational Research Society*, 53(8):895–906, August 2002.

[HB06] Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization*. The MIT Press, 2006.

[HB10] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, November 2010.

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523, Berlin, Heidelberg, 2011. Springer.

[HHLS09] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[HJS+18] Marijn JH Heule, Matti Juhani Järvisalo, Martin Suda, et al. Proceedings of sat competition 2018: Solver and benchmark descriptions. 2018.

[HK00] Sönke Hartmann and Rainer Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127(2):394–407, December 2000.

[HL05] Willy Herroelen and Roel Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, 165(2):289–306, September 2005.

[HM14] Pierre Hansen and Nenad Mladenović. Variable Neighborhood Search. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 313–337. Springer US, Boston, MA, 2014.

[HMB+14] Iiro Harjunkoski, Christos T. Maravelias, Peter Bongers, Pedro M. Castro, Sebastian Engell, Ignacio E. Grossmann, John Hooker, Carlos Méndez, Guido Sand, and John Wassick. Scope for industrial applications of production scheduling models and solution methods. *Computers & Chemical Engineering*, 62:161–193, March 2014.

[HW06] Canberra Hyatt and Patrick Weaver. A brief history of scheduling. *Melbourne, Australia: Mosaic Project Services Pty Ltd*, 2006.

[HZC+19] Zizhao Huang, Zilong Zhuang, Qi Cao, Zhiyao Lu, Liangxun Guo, and Wei Qin. A survey of intelligent algorithms for open shop scheduling problem. *Procedia CIRP*, 83:569–574, 2019.

[JRL08] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: An open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

[Kar84] Narendra Karmarkar. A New Polynomial-Time Algorithm for Linear Programming-II. *Combinatorica*, 4:373–395, December 1984.

[KAS79] T. KASHIWABARA. NP-completeness of the problem of finding a minimal-cliquenumber interval graph containing a given graph as a subgraph. *Proc. 1979 Int. Symp. Circuit Syst*, pages 657–660, 1979.

[KAS08] Madjid Khichane, Patrick Albert, and Christine Solnon. Integration of ACO in a Constraint Programming Language. In *Ant Colony Optimization and Swarm Intelligence*, Lecture Notes in Computer Science, pages 84–95, Berlin, Heidelberg, 2008. Springer.

[KAS10] Madjid Khichane, Patrick Albert, and Christine Solnon. Strong Combination of Ant Colony Optimization with Constraint Programming Optimization. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140, pages 232–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[KCEP08] Georgios M. Kopanos, Elisabet Capón-García, Antonio Espuña, and Luis Puigjaner. Costs for Rescheduling Actions: A Critical Issue for Reducing the Gap between Scheduling Theory and Practice. *Industrial & Engineering Chemistry Research*, 47(22):8785–8795, November 2008.

[KH14] Jihene Kaabi and Youssef Harrath. A Survey of parallel machine scheduling under availability constraints. *International Journal of Computer and Information Technology*, 3:238–245, March 2014.

[Kha80] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, January 1980.

[KHNT18] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, November 2018.

[KKJC02] Dong-Won Kim, Kyong-Hee Kim, Wooseung Jang, and Frank F Chen. Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer-Integrated Manufacturing*, 18(3-4):223–231, June 2002.

[Kot14] Lars Kotthoff. LLAMA: Leveraging Learning to Automatically Manage Algorithms. *arXiv:1306.1031 [cs]*, April 2014.

[Kot16] Lars Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. In *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*, Lecture Notes in Computer Science, pages 149–190. Springer International Publishing, Cham, 2016.

[LBG91] Manuel Laguna, J. Wesley Barnes, and Fred W. Glover. Tabu search methods for a single machine scheduling problem. *Journal of Intelligent Manufacturing*, 2(2):63–73, April 1991.

[LG07] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris*, 8, 2007.

[Liu09] Ming Liu. *Design and Evaluation of Algorithms for Online Machine Scheduling Problems*. PhD thesis, Ecole Centrale Paris, 2009.

[LLSX17] Guo Li, Mengqi Liu, Suresh P. Sethi, and Dehua Xu. Parallel-machine scheduling with machine-dependent maintenance periodic recycles. *International Journal of Production Economics*, 186:1–7, April 2017.

[Lou03] Robin Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[LP97] Young Hoon Lee and Michael Pinedo. Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research*, 100(3):464–474, August 1997.

[LPT97] C.-Y. Lee, S. Piramuthu, and Y.-K. Tsai. Job shop scheduling with a genetic algorithm and machine learning. *International Journal of Production Research*, 35(4):1171–1191, April 1997.

[LR08] Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *FLAIRS Conference*, pages 555–560, 2008.

[LRSV18] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23(2):210–250, April 2018.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, September 1993.

[LW66] E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, August 1966.

[LXCZ09] Ming Liu, Yinfeng Xu, Chengbin Chu, and Feifeng Zheng. Online scheduling to minimize modified total tardiness with an availability constraint. *Theoretical Computer Science*, 410(47):5039–5046, November 2009.

[LY02] Alexandre Linhares and Horacio Hideki Yanasse. Connections between cutting-pattern sequencing, VLSI design, and exible machines. *Operations Research*, page 14, 2002.

[LYL13] Jae-Ho Lee, Jae-Min Yu, and Dong-Ho Lee. A tabu search algorithm for unrelated parallel machine scheduling with sequence- and machine-dependent setups: Minimizing total tardiness. *The International Journal of Advanced Manufacturing Technology*, 69(9-12):2081–2089, December 2013.

[Man87a] CP Optimizer User's Manual. Ibm ilog CP optimizer. *Version*, Version 12, release 8:1987–2018, 1987.

[Man87b] CPLEX User's Manual. Ibm ilog cplex optimization studio. *Version*, 12:1987–2018, 1987.

[MCZ10] Ying Ma, Chengbin Chu, and Chunrong Zuo. A survey of scheduling with deterministic machine availability constraints. *Computers & Industrial Engineering*, 58(2):199–211, March 2010.

[MDC14] Tommy Messelis and Patrick De Causmaecker. An automatic algorithm selection approach for the multi-mode resource-constrained project scheduling problem. *European Journal of Operational Research*, 233(3):511–528, March 2014.

[ME04] Bernd Meyer and Andreas Ernst. Integrating ACO and Constraint Propagation. In *Ant Colony Optimization and Swarm Intelligence*, Lecture Notes in Computer Science, pages 166–177, Berlin, Heidelberg, 2004. Springer.

[Mey08] Bernd Meyer. Hybrids of Constructive Metaheuristics and Constraint Programming: A Case Study with ACO. In *Hybrid Metaheuristics*, volume 114, pages 151–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[MK94] Min Hee Kim and Yeong-Dae Kim. Simulation-based real-time scheduling in a flexible manufacturing system. *Journal of Manufacturing Systems*, 13(2):85–93, January 1994.

[Mok04] Ethel Mokotoff. An exact algorithm for the identical parallel machine scheduling problem. *European Journal of Operational Research*, page 12, 2004.

[Mol05] Balázs Molnár. Multi-criteria scheduling of order picking processes with simulation optimization. *Periodica Polytechnica Transportation Engineering*, 33(1-2):59–68, 2005.

[MPE03] Kostas S. Metaxiotis, John E. Psarras, and Kostas A. Ergazakis. Production scheduling in ERP systems: An AI-based approach to face the gap. *Business Process Management Journal*, 9(2):221–247, January 2003.

[MSCK09] Racem Mellouli, Chérif Sadfi, Chengbin Chu, and Imed Kacem. Identical parallel-machine scheduling under availability constraints to minimize the sum of completion times. *European Journal of Operational Research*, 197(3):1150–1165, September 2009.

[MSMA19] A. Muñoz-Villamizar, J. Santos, J. Montoya-Torres, and M. Alvaréz. Improving effectiveness of parallel machine scheduling with earliness and tardiness costs: A case study. *International Journal of Industrial Engineering Computations*, 10(3):375–392, 2019.

[MSV10] Vittorio Maniezzo, Thomas Stützle, and Stefan Voß. *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*, volume 10 of *Annals of Information Systems*. Springer US, 2010.

[Mur83] Katta G. Murty. *Linear Programming*. Wiley, New York, 1983.

[NH10] Juan M. Novas and Gabriela P. Henning. Reactive scheduling framework based on domain knowledge and constraint programming. *Computers & Chemical Engineering*, 34(12):2129–2148, December 2010.

[NMJ09] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-Based Selection of Policies for SAT Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 326–340, Berlin, Heidelberg, 2009. Springer.

[NS03] Klaus Neumann and Christoph Schwindt. Project scheduling with inventory constraints. *Mathematical Methods of Operations Research (ZOR)*, 56(3):513–533, January 2003.

[NSBL18] Vitor Nesello, Anand Subramanian, Maria Battarra, and Gilbert Laporte. Exact solution of the single-machine scheduling problem with periodic maintenances and sequence-dependent setup times. *European Journal of Operational Research*, 266(2):498–507, April 2018.

[OP08] Djamila Ouelhadj and Sanja Petrovic. A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, 12(4):417, October 2008.

[OQ13] Pierre Ouellet and Claude-Guy Quimper. Time-Table Extended-Edge-Finding for the Cumulative Constraint. In *Principles and Practice of Constraint Programming*, volume 8124, pages 562–577. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[PF17] Michele E. Pfund and John W. Fowler. Extending the boundaries between scheduling and dispatching: Hedging and rescheduling techniques. *International Journal of Production Research*, 55(11):3294–3307, June 2017.

[PF19] Laurent Perron and Vincent Furnon. OR-Tools, October 2019.

[PFG04] Michele Pfund, John W. Fowler, and Jatinder N. D. Gupta. A Survey of Algorithms for Single and Multi-Objective Unrelated Parallel-Machine Deterministic Scheduling Problems. *Journal of the Chinese Institute of Industrial Engineers*, 21(3):230–241, January 2004.

[PGGM13] Victor Pillac, Michel Gendreau, Christelle Guéret, and Andrés L. Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, February 2013.

[PGM12a] Victor Pillac, Christelle Guéret, and Andrés Medaglia. *A Fast Re-Optimization Approach for Dynamic Vehicle Routing*. PhD thesis, Ecole des Mines de Nantes, 2012.

[PGM12b] Victor Pillac, Christelle Guéret, and Andrés Medaglia. On the Dynamic Technician Routing and Scheduling Problem. Research Report, Ecole des Mines de Nantes, 2012.

[Pin16] Michael L. Pinedo. *Scheduling*. Springer International Publishing, Cham, 2016.

[PPR18] Adriana Pacheco, Cédric Pralet, and Stéphanie Roussel. Techniques de décisions hiérarchiques pour l'ordonnancement de tâches. In *JIAF + JFPC 2018*, AMIENS, France, June 2018.

[PPR19a] Adriana Pacheco, Cédric Pralet, and Stéphanie Roussel. Constraint based scheduling with complex setup operations: An iterative two-layer approach. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1155–1161. International Joint Conferences on Artificial Intelligence Organization, July 2019.

[PPR19b] Adriana Pacheco, Cédric Pralet, and Stéphanie Roussel. Decomposition and Cut Generation Strategies for Solving Multi-Robot Deployment Problems. In *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 472–487, Cham, 2019. Springer International Publishing.

[PR10] David Pisinger and Stefan Ropke. Large Neighborhood Search. In *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 399–419. Springer US, Boston, MA, 2010.

[PST04] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. 2004.

[Ric76] John R. Rice. The Algorithm Selection Problem. In *Advances in Computers*, volume 15, pages 65–118. Elsevier, January 1976.

[RTF18] Daniel Alejandro Rossit, Fernando Tohmé, and Mariano Frutos. The Non-Permutation Flow-Shop scheduling problem: A literature review. *Omega*, 77:143–153, June 2018.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[Sal02] S. Salhi. Defining tabu list size and aspiration criterion within tabu search methods. *Computers & Operations Research*, 29(1):67–86, January 2002.

[SB99] R. Shafaei and P. Brunn. Workshop scheduling using practical (inaccurate) data Part 1: The performance of heuristic scheduling rules in a dynamic job shop environment using a rolling time horizon approach. *International Journal of Production Research*, 37(17):3913–3925, November 1999.

[Sch96] J.M.J. Schutten. List scheduling revisited. *Operations Research Letters*, 18(4):167–170, February 1996.

[SD08] S.N. Sivanandam and S.N. Deepa. Genetic Algorithms. In *Introduction to Genetic Algorithms*, pages 15–37. Springer, Berlin, Heidelberg, 2008.

[SFRS20] Zachariah Stevenson, Ricardo Fukasawa, and Luis Ricardez Sandoval. Evaluating periodic rescheduling policies using a rolling horizon framework in an industrial-scale multipurpose plant. *Journal of Scheduling*, 23(3):397–410, June 2020.

[SH98] T. Stützle and H. Hoos. *Improvements on the Ant-System: Introducing the MAX-MIN Ant System*, pages 245–249. Springer Vienna, Vienna, 1998.

[Sha98] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Principles and Practice of Constraint Programming — CP98*, Lecture Notes in Computer Science, pages 417–431, Berlin, Heidelberg, 1998. Springer.

[SLT06] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www. gecode. org*, pages 11–13, 2006.

[Sol10] Christine Solnon. *Ant Colony Optimization and Constraint Programming*. John Wiley & Sons, Ltd, 2010.

[Stu70] L. B. J. M. Sturm. A Simple Optimality Proof of Moore's Sequencing Algorithm. *Management Science*, 17(1):116–118, 1970.

[SW96] Paul P.M. Stoop and Vincent C.S. Wiers. The complexity of scheduling in practice. *International Journal of Operations & Production Management*, 16(10):37–53, January 1996.

[TNGF13] R.F. Tavares Neto and M. Godinho Filho. Literature review regarding Ant Colony Optimization applied to scheduling problems: Guidelines for implementation and directions for future research. *Engineering Applications of Artificial Intelligence*, 26(1):150–161, January 2013.

[V+15] Robert J Vanderbei et al. *Linear Programming*, volume 3. Springer, 2015.

[VM87] Ari P. J. Vepsalainen and Thomas E. Morton. Priority Rules for Job Shops with Weighted Tardiness Costs. *Management Science*, 33(8):1035–1047, August 1987.

[Wal99] Toby Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1172–1177, San Francisco, CA, USA, July 1999. Morgan Kaufmann Publishers Inc.

[Wie97] Vincent CS Wiers. *Human Computer Interaction in Production Scheduling.* Citeseer, 1997.

[WWY+18] Shijin Wang, Xiaodong Wang, Jianbo Yu, Shuan Ma, and Ming Liu. Bi-objective identical parallel machine scheduling to minimize total energy consumption and makespan. *Journal of Cleaner Production*, 193:424–440, August 2018.

[XHHL08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, July 2008.

[Yan97] Horacio Hideki Yanasse. On a pattern sequencing problem to minimize the maximum number of open stacks. *European Journal of Operational Research*, page 10, 1997.

FOLIO ADMINISTRATIF

THÈSE DE L'UNIVERSITÉ DE LYON OPÉRÉE AU SEIN DE L'INSA LYON

Nom : Groleaz
Prénom : Lucas

Date de soutenance : 07-06-2021

Titre : The Group Cumulative Scheduling Problem

Nature : Doctorat
Numéro d'ordre :
École doctorale : InfoMaths
Spécialité : Informatique

Résumé :

La société Infologic développe un ERP, appelé Copilote, spécialisé pour les entreprises du secteur agro-alimentaire. Il intègre plusieurs modules permettant d'ordonnancer différentes opérations de la chaîne de production. Ces modules apportent des solutions à différents problèmes d'ordonnancement ayant des contraintes et des objectifs différents. Par ailleurs, bien que la littérature concernant les problèmes d'ordonnancement soit vaste, une contrainte particulière rencontrée par les utilisateurs de Copilote ne peut que difficilement être modélisée en utilisant les éléments connus de la littérature. Dans le problème rencontré, les opérations à ordonnancer sont réparties en groupes. L'ordonnancement doit satisfaire une contrainte sur ces groupes assurant qu'à tout moment il n'y a pas plus de k groupes pour lesquels des opérations ont été commencées tandis que d'autres ne sont pas terminées. Dans cette thèse, nous étudions ce nouveau problème d'ordonnancement d'un point de vue théorique, et nous proposons des adaptations pour les méthodes de résolution classiquement utilisées pour les problèmes d'ordonnancement (programmation linéaire en nombres entiers, programmation par contraintes, optimisation par colonies de fourmis, et recherche locale). Nous introduisons également une nouvelle approche hybridant programmation par contraintes et optimisation par colonies de fourmis pour résoudre ce problème. Nous comparons expérimentalement ces différents algorithmes sur un jeu d'essai construit à partir de données réelles, et nous montrons que le meilleur algorithme change en fonction des caractéristiques de l'instance à résoudre. Nous proposons donc une méthode, qui, selon les caractéristiques de l'instance à résoudre, choisit automatiquement la méthode de résolution la plus adaptée. Finalement, nous évaluons, dans un contexte dynamique, le coût engendré par le fait de perturber le moins possible les plannings déjà établis lorsque de nouvelles données sont révélées.

Mots-clefs : Scheduling - Cumulative Constraint - Metaheursitics - Constraint Programming - Algorithm Selection

Laboratoire de recherche : LIRIS

Directrice de thèse : Christine Solnon
Président de jury :
Composition du jury :

Christian Artigues - Yves Deville - Nadia Brauner
Philippe Laborie - Christine Solnon - Samba Ndojh Ndiaye