Introduction à la Recherche en Laboratoire

Ensimag - G-Scop

Analyse comparative de solveurs pour le problème d'ordonnancement $1 \mid r_j, \tilde{d}_j \mid \sum w_j T_j$

Auteur :
Léa Petit-Jean Genat

Encadrant :
Nadia Brauner

Co-endradrant :
Cléophée Robin

Février - Mai 2022

Remerciements

Je voudrais tout d'abord adresser toute ma gratitude à mes deux encadrantes. Nadia Brauner pour sa confiance, sa disponibilité et surtout pour avoir su partager ses connaissances, et Cléophée Robin pour son intérêt, ses interrogations et ses retours réguliers qui m'ont permis de m'interroger et d'améliorer ce compte-rendu.

Ce travail ne serait pas aussi complet sans l'aide d'Hadrien Cambazard. Je le remercie, d'une part de m'avoir grandement aidé à la compréhension et à la prise en main du solveur CPO et d'avoir vérifié mon modèle. Et d'autre part, d'avoir pris le temps d'échanger avec moi et de m'avoir donné des pistes d'interprétation et d'approfondissement des résultats obtenus.

Un grand merci aussi à Julien Darlay d'avoir répondu à mes questions sur le fonctionnement du solveur LocalSolver, d'avoir vérifié mon modèle et de m'avoir guidé dans le choix d'une méthode de comparaison des solveurs.

Et enfin, je tiens à remercier Zoltán Szigeti pour m'avoir permis de réaliser ce stage d'introduction à la recherche et le laboratoire G-SCOP de m'avoir accueilli.

Table des matières

Présentation du stage

Ce document est le compte-rendu d'un stage d'introduction à la recherche en laboratoire et a pour but la restitution du travail effectué. Le stage est réalisé par Léa Petit-Jean Genat et encadré par Nadia Brauner en collaboration avec Cléophée Robin et se déroule au laboratoire G-SCOP au rythme d'un après-midi par semaine, dans le cadre du module IRL de l'Ensimag. Le laboratoire est pluridisciplinaire, autour de problématiques de recherche relatives à la conception, l'optimisation et la gestion des produits et des systèmes de production.

Ce travail s'inscrit dans l'équipe Recherche Opérationnelle pour les Systèmes de Production (ROSP). Les travaux de cette équipe, modèles théoriques ou applications industrielles, portent sur l'aide à la prise de décision pour optimiser, grâce à des outils et techniques de recherche opérationnelle, les performances des systèmes de production, que ce soit, une simple machine sur une ligne de production, un atelier complet, etc.

Le sujet traite d'un problème d'ordonnancement, noté $1 \,|\, r_j, \tilde{d}_j \,|\, \sum w_j T_j$. Ce problème a été choisi assez difficile afin de pouvoir comparer l'efficacité des différentes modélisations proposées en fonction des solveurs choisis pour la résolution d'instances du problème. Un solveur est un logiciel qui permet de calculer et fournir le résultat d'un problème après sa transcription informatique. L'objectif principal est de prendre en main, puis comparer, plusieurs solveurs. Les résultats montreront quels solveurs semblent plus efficaces pour résoudre des instances de ce problème d'ordonnancement.

Comparer des solveurs et des modélisations pour un problème particulier est utile pour les chercheurs, ingénieurs et industriels. Les travaux de ce type, par exemple ceux de Roselli, Bengtsson et Åkesson (2018) qui comparent deux types de solveurs pour un problème d'ordonnancement, permettent de donner des points de référence pour classer les solveurs et sert souvent pour faire une comparaison qualité-prix.

Il est important de noter le rôle des analyses comparatives (*benchmarks*) dans le développement des solveurs. Ces références sont essentielles pour adapter les solveurs à diverses classes d'instances pratiques, c'est-à-dire générées par des applications du monde réel (Malik et Lintao, 2009).

Dans la section 1, nous présenterons globalement les problèmes d'ordonnancement et l'état de l'art. Dans la section 2, nous détaillerons le problème $1 \,|\, r_j, \tilde{d}_j \,|\, \sum w_j T_j$, c'est-à-dire : sa description et sa complexité. Nous verrons également dans cette section, comment générer des instances du problème et quels solveurs seront utilisés pour les résoudre. Dans la section 3, nous montrerons et expliquerons les modèles réalisés. Enfin la section 4 est entièrement consacrée aux solveurs, de l'installation jusqu'aux résultats obtenus, en passant par une comparaison de prise en main.

1 Introduction à l'ordonnancement

La théorie de l'**ordonnancement** (Leung, 2004) est une branche de la recherche opérationnelle qui s'intéresse au calcul de dates d'exécution optimales de tâches. Une tâche possède une durée et nécessite un certains nombre de ressources pour être effectuée. Une ressource est un moyen, technique ou humain dont la disponibilité est connue *a priori*. L'ordre optimal d'exécution des tâches peut varier selon le critère à privilégier (minimiser le temps d'exécution total ou minimiser les retards des tâches, par exemple), les ressources disponibles (une ou plusieurs), et les contraintes imposées (temporelles ou de ressources).

Les problèmes d'ordonnancement sont présents dans de nombreux domaines tels que l'informatique (organisation des programmes), la construction (suivi de projet), l'industrie (ateliers) ou encore l'administration (emplois du temps), d'après Gotha (1993). Ils peuvent être très variés, c'est pourquoi une notation particulière pour les problèmes d'ordonnancement existe. Par exemple $1|r_j, \tilde{d}_j$ désigne un problème à une machine avec des dates d'arrivée et des *deadlines*. Cette notation, présente dans « Definition, Analysis and Classification of Scheduling Problems » (2007), permet de savoir plus facilement et rapidement le type de problème d'ordonnancement dont il est question.

Les problèmes d'ordonnancement les plus étudiés dans la littérature ont souvent pour objectif la minimisation de la date de fin globale notée $C_{\max}$ sur plusieurs machines. Le problème *Job Shop* (JSP) est un des plus célèbres et il est possible de trouver de nombreux travaux de recherche avec des méthodes de résolution différentes comme les travaux de Cheng, Gen et Tsujimura (1996) avec des algorithmes génétiques (algorithmes d'optimisation, reposant le principe de la sélection naturelle) ou encore des travaux de Ku et Beck (2016) avec la *programmation linéaire en nombres entiers* ou *Mixed Integer Programming* (MIP), méthode que nous détaillerons dans la section 2.2 pour notre problème, etc. Les contraintes du problème *Job Shop* sont très générales : aucune tâche ne peut commencer avant que la précédente soit terminée, une machine ne peut travailler que sur une tâche à la fois et une fois une tâche commencée, elle doit impérativement être finie. Il est fréquent de voir aussi le problème de *Flow Shop* qui est un cas particulier de *Job Shop* mais avec un ordre d'opérations imposé sur les différentes machines, comme par exemple dans les travaux de KING et SPACHIS (1980). Un dernier type de problème très commun est le problème RCPSP (*Resource Constrained Project Scheduling Problems*). Ce problème correspond à minimiser la date de fin $C_{\max}$ d'un projet sachant que les ressources ont des disponibilités limitées et les tâches possèdent des contraintes de précédence, c'est-à-dire qu'elles ne peuvent pas commencer avant d'autres tâches.

D'autres objectifs, souvent étudiés également sont : la minimisation du nombre U_j (pondéré par des poids w_j) de tâches en retard $\sum_j (w_j)U_j$ (E. L. Lawler et Moore, 1969), la minimisation du retard T_j (pondéré) des tâches $\sum_j (w_j)T_j$ (E. Lawler, 1977) ou la minimisation des dates de fin C_j (pondérées) de chaque tâche $\sum_j (w_j)C_j$ (Hoogeveen, Schuurman et Woeginger, 2001). Ces objectifs sont souvent accompagnés de contraintes telles que des dates d'arrivées pour chaque tâche r_j (Skutella, 2006) ou des contraintes de précédence *prec* (Chekuri et Motwani, 1999).

Le problème d'ordonnancement qui nous intéresse est décrit dans la prochaine section.

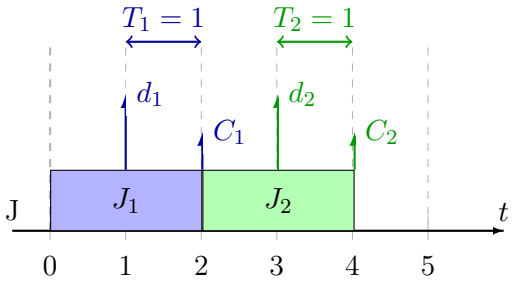
2 Description du problème d'ordonnancement $1 \,|\, \mathbf{r_j}, \tilde{\mathbf{d}}_\mathbf{j} \,|\, \sum \mathbf{w_j T_j}$

Le problème $\mathbf{1} \,|\, \mathbf{r_j}, \tilde{\mathbf{d}}_\mathbf{j} \,|\, \sum \mathbf{w_j T_j}$ correspond à la description suivante : une unique machine doit exécuter des tâches (*jobs*) J_j, $j = 1, 2, \ldots, n$. Toutes les tâches doivent être exécutées sur la machine et respecter leurs dates limites (*deadlines*) respectives. Chaque tâche possède une date de disponibilité (*release date*) r_j, une *deadline* $\tilde{d}_j$, une durée (*processing time*) p_j, une date échue (*due date*) d_j ainsi qu'un poids positif (*weight*) w_j. Une tâche ne peut pas commencer avant la date d'arrivée. La *due date* correspond à la date où la tâche devrait être terminée et à partir de laquelle le retard de travail (*tardiness*) est compté. La différence entre la date échue et la *deadline* est le fait qu'une tâche peut se terminer après la date échue, même si elle coûtera plus cher à la réalisation, alors qu'elle ne peut pas se terminer après la *deadline*. (La date échue peut être inférieure à la date d'arrivée plus la durée de la tâche). Pour un ordre donné, la date de fin d'une tâche (*completion time*) est notée C_j et le retard de travail est noté $T_j = \max(0,\, C_j - d_j)$. L'objectif est de minimiser la somme pondérée des retards de travail *i.e.* $\sum_j w_j T_j$.

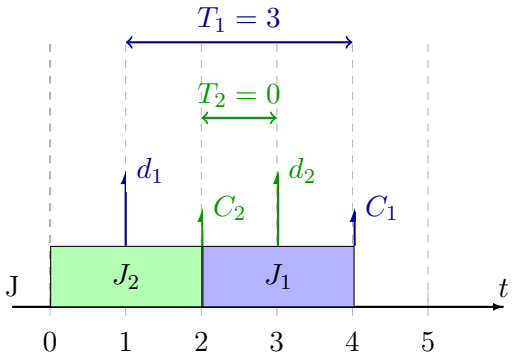
Par exemple, soient deux tâches à exécuter sur la machine :

	J_1	J_2
r_j	0	0
$\tilde{d}_j$	4	4
p_j	2	2
d_j	1	3
w_j	3	1

Deux solutions sont réalisables :



Valeur de l'objectif : $w_1 T_1 + w_2 T_2 = 3 + 1 = \boxed{4}$



Valeur de l'objectif : $w_1T_1 + w_2T_2 = 3 \times 3 = \boxed{9}$

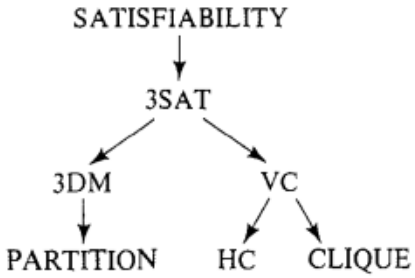
La première solution est meilleure puisque l'objectif est de minimiser et $4 < 9$.

D'après Roselli, Bengtsson et Åkesson (2018), les problèmes d'ordonnancement sont souvent des problèmes $\mathcal{NP}$-difficile.

En informatique théorique Garey et Johnson (1978) montrent qu'un problème $\mathcal{NP}$-difficile est un problème vers lequel il est possible de ramener tout problème de la classe $\mathcal{NP}$ (non déterministe polynomial) par une réduction polynomiale.

Il faut savoir qu'il existe une hiérarchie des problèmes et qu'il est possible de réduire tous les problèmes $\mathcal{NP}$-complet au problème SAT (*Boolean Satisfiability* ou satifaisabilité booléenne en français).

Les six problèmes ci-dessous ont été prouvé $\mathcal{NP}$-complet par Garey et Johnson (1978). Les flèches représentent les réductions. Par exemple, 3SAT se réduit à SAT signifie qu'il existe une transformation polynomiale d'une instance quelconque du problème 3SAT à une instance du problème SAT. Cet exemple est facile à comprendre puisque le problème SAT est plus difficile que la version restreinte de lui-même.



Théorème 2.1. *Le problème* $\mathbf{1 \,|\, r_j, \tilde{d}_j \,|\, \sum w_j T_j}$ *est $\mathcal{NP}$-complet.*

Démonstration. Prouvons qu'il existe une réduction polynomiale entre les deux problèmes de décision ci-dessous :

$1|r_j\tilde{d}_j$ (*Sequencing within intervals*)

INSTANCE : Ensemble fini J de tâches et pour chaque tâche $j \in J$, une date d'arrivée $r_j \in \mathbb{N}$, une durée $p_j \in \mathbb{N}^+$ et une *deadline* $\tilde{d}_j \in \mathbb{N}^+$.

QUESTION : Existe-t-il un ordonnancement réalisable pour J, c'est-à-dire existe-t-il une date de début effective $s_j \in \mathbb{N}$ pour chaque tâche $j \in J$ telle que $s_j \geq r_j$ et $s_j + p_j \leq \tilde{d}_j$ sachant que deux tâches ne peuvent pas être exécutées en même temps $s_j \notin [\![s_i, s_i + p_i[\![\quad \forall i, j \in J$

PARTITION (Problème $\mathcal{NP}$-complet)

INSTANCE : Ensemble fini A d'entiers.

QUESTION : Existe-t-il une partition $\{A_1, A_2\}$ de A avec $S = \sum_{a \in A} a$ telle que

$$\sum_{a \in A_1} a = \frac{S}{2} = \sum_{a \in A_2} a$$

Transformons une instance du problème PARTITION en une instance du problème $1|r_j\tilde{d}_j$:

Pour chaque entier $a \in A$, on définit une tâche $j = \{r_j, p_j, \tilde{d}_j\}$ avec $r_j = 0$ la date d'arrivée, $p_j = a$ la durée et $\tilde{d}_j = S + 1$ la *deadline*.

Ajoutons une tâche fictive $f = \{r_f, p_f, \tilde{d}_f\}$ avec $r_f = \lceil \frac{S}{2} \rceil$ la date d'arrivée, $p_f = 1$ la durée et $\tilde{d}_f = \lceil \frac{S+1}{2} \rceil$ la *deadline*.

1. **S impaire :**
   Dans le cas où la somme est impaire, aucun ordonnancement n'est réalisable car $r_f = \lceil \frac{S}{2} \rceil = \lceil \frac{S+1}{2} \rceil = \tilde{d}_f$. Or, $p_f = 1 > \tilde{d}_f - r_f = 0$, donc la tâche fictive f ne peut pas être effectuée.
   Cela fait bien le lien avec le problème PARTITION, dans lequel il est impossible d'obtenir une partition si la somme de tous les entiers est impaire car

   $$\frac{S}{2} \neq \sum_{a \in A_1} a \neq \sum_{a \in A_2} a$$

   Donc, il n'existe pas d'ordonnancement réalisable et le problème PARTITION n'est pas réalisable.
2. **S paire :**
   Dans le cas où la somme est paire, la tâche fictive f doit obligatoirement avoir $s_f = \frac{S}{2}$ comme date de début effective pour qu'un ordonnancement soit réalisable. Placer cette tâche fictive divise le temps disponible pour réaliser l'ordonnancement en deux blocs de taille $\frac{S}{2}$, AVANT et APRÈS la tâche fictive. Nous savons que chaque bloc sera rempli puisque le temps total disponible $2 \times \frac{S}{2}$ est égal à la somme S des durées des tâches restantes à placer dans l'ordonnancement.

   - Si nous trouvons un ordonnancement réalisable, nous obtenons un ensemble de tâches exécutées AVANT la tâche fictive f et un autre ensemble de tâches exécutées APRÈS f, ce qui revient à une PARTITION de tâches non fictives $\{J_1, J_2\}$ avec $J_1 = \{j \in J : j \neq f \text{ et } s_j + p_j \leq b_f\}$ et $J_2 = \{j \in J : j \neq f \text{ et } s_j > b_f\}$.
     $\{J_1, J_2\}$ définit une partition réalisable pour le problème PARTITION :

     $$\sum_{j \in J_1} p_j = \sum_{j \in J_2} p_j = \frac{1}{2} \sum_{a \in A} a$$

- Si nous trouvons une PARTITION $\{A_1, A_2\}$ d'entiers, alors, nous pouvons construire un ordonnancement de sorte que tous les entiers de A_1 soit exécutée AVANT la tâche fictive et tous les entiers de A_2 soit exécutée APRÈS la tâche fictive. En effet, d'après la construction d'une instance pour le problème d'ordonnancement, $r_j = 0 \leq s_j \leq S+1 = \tilde{d}_j \quad \forall j \in J$. Rien n'empêche donc aux tâches construites à partir de A_1 de finir plus tôt ou aux tâches construites à partir de A_2 de commencer plus tard que la tâche fictive f. L'ordonnancement obtenu est bien réalisable puisque, pour toutes les tâches, $p_a = a \quad \forall a \in A$ et le temps total disponible est $2\sum \frac{S}{2}$.

La transformation entre les deux problèmes est polynomiale et proportionnelle au nombre de tâches, puisqu'il suffit de définir une tâche $j = \{r_j = 0, p_j = a, \tilde{d}_j = S+1\}$ pour chaque entier a du problème PARTITION plus une tâche fictive $f = \{r_f = \lceil \frac{S}{2} \rceil, p_f = 1, \tilde{d}_f = \lceil \frac{S+1}{2} \rceil\}$. Le problème PARTITION étant $\mathcal{NP}$-complet, cela implique que le problème $\mathbf{1|r_j\tilde{d}_j}$ est $\mathcal{NP}$-complet.

Le problème $1 \,|\, r_j, \tilde{d}_j \,|\, \sum w_j T_j$ est plus difficile puisqu'en plus de chercher un ordonnancement réalisable, il cherche à optimiser une fonction objectif qui correspond à la somme pondérée des retards des tâches. Si nous savons résoudre le problème d'optimisation, alors il nous suffira de récupérer l'ordonnancement trouvé, qui sera réalisable. Cette récupération s'effectue en temps polynomial car il faut simplement récupérer les dates de début effectives trouvées. Par conséquent, $\mathbf{1 \,|\, r_j, \tilde{d}_j \,|\, \sum w_j T_j}$ est $\mathcal{NP}$-complet. □

2.1 Génération des instances

Pour tester les solveurs, il faut générer des instances du problème. Les instances doivent pouvoir être de taille différente et réalisable pour que les solveurs puissent trouver une solution. L'idée pour générer des instances de tests, est de générer un ordonnancement réalisable, c'est-à-dire un enchaînement des tâches avec respect des durées et des *deadlines*, et de faire varier les dates d'arrivées et les dates échues aléatoirement, selon une loi uniforme, comme tous les tirages à suivre. Les instances sont pseudo-aléatoire puisque châque tâche dépend de celle d'avant en ce qui concerne sa *deadline*. Pour chaque tâche, la durée de cette dernière est choisie entre 1 et une borne supérieure arbitraire. Ensuite, la *deadline* est choisie entre une borne inférieure (*deadline* de la tâche précédente) plus la durée choisie, et la borne inférieure plus la borne supérieure (durée maximum tirée précedement). La date d'arrivée est ensuite tirée entre 0 et le début au plus tard de la tâche j, c'est-à-dire $\tilde{d}_j - p_j$. La date échue est quant à elle tirée entre 0 et la *deadline*. Le poids, lui, est tiré entre 1 et 100. Après la génération d'une tâche, il faut mettre à jour la borne inférieure. A la fin, après avoir généré toutes les tâches, un mélange est effectué afin de ne pas préserver l'ordre de construction.

Une instance est de la forme :

```
n = 2;
r = [0, 0];
deadline = [4, 4];
p = [2, 2];
d = [1, 3];
w = [3, 1];
```

Avec n correspondant au nombre de tâches, r correspondant à la date d'arrivée, p correspondant à la durée, d correspondant à la date échue, *deadline*, comme son nom l'indique, correspondant à la *deadline* et w correspondant au poids.

2.2 Solveurs utilisés pour la résolution du problème

Chaque solveur (logiciel) est développé différemment et par conséquent, chaque transcription informatique du problème est différente et n'utilise pas forcément la même logique ou méthode de résolution.

Pour les problèmes d'ordonnancement, il est question, d'après le papier de Ku et Beck (2016), de solveurs MIP - *Mixed Integer Programming* (*Programmation Linéaire en Nombres Entiers* (PLNE) en français), CP - *Constraint Programming* (*Programmation par Contraintes*), SAT - *Boolean Satisfiability* (satisfaisabilité booléenne) ou encore LS - *Local Search* (recherche locale).

Un modèle MIP est un modèle avec des contraintes linéaires dans lequel certaines variables de décision sont entières (*Integer Programming* (IP) ou *programmation linéaire*) et dont certaines sont binaires (de valeur 0 ou 1). Cela permet de modéliser plus de problèmes d'optimisation que la *programmation linéaire*. Cependant, la résolution d'un MIP est $\mathcal{NP}$-difficile. Les solveurs MIP sont souvent basés sur les techniques de *programmation linéaire* et de *branch and bound*. Le *branch and bound* ou séparation et évaluation en français, est un algorithme qui consiste à créer et parcourir un arbre d'exploration de sous-problèmes et de vérifier s'il faut poursuivre l'exploration ou non en fonction de bornes, supérieure et inférieure, calculées lors du parcours de l'arbre.

La *programmation par contraintes* (CP) permet de résoudre des problèmes combinatoires de grande taille tels que des problèmes d'ordonnancement. Dans le cadre de la *programmation par contraintes*, les problèmes sont modélisés à l'aide de variables de décision et de contraintes, où une contrainte est une relation entre une ou plusieurs variables qui limite les valeurs que peuvent prendre simultanément chacune des variables liées par la contrainte (Pinedo, 2012). Les solveurs CP consistent à réduire, pas à pas, le domaine d'une variable afin de maintenir l'ensemble des valeurs possibles cohérent avec les contraintes du problème. Ils utilisent la propagation de contraintes, c'est-à-dire que le choix effectué pour une variable est répercuté sur le domaine des autres variables. Souvent, un algorithme de recherche arborescente est utilisé (Bessière et al., 2005).

Le problème de satisfaisabilité booléenne (SAT) est un problème avec des contraintes booléennes à satisfaire (Malik et Lintao, 2009). Ce problème est $\mathcal{NP}$-complet et les algorithmes utilisés pour le résoudre sont appelés solveurs SAT. Ce type de solveur est souvent basé sur des heuristiques de recherche locale (Stuckey, 2012). Une heuristique est, d'après Müller-Merbach (1981), une technique conçue pour résoudre un problème plus rapidement lorsque les méthodes classiques sont trop lentes, ou pour trouver une solution approximative lorsque les méthodes classiques ne trouvent pas de solution exacte.

La recherche locale (LS) est une méthode pour résoudre des problèmes d'optimisation et consiste à passer d'une solution à une autre solution, proche dans l'espace de recherche (Ruiz et Pranzo, 2006). La plupart des algorithmes de recherche locale sont *anytime*, c'est-à-dire qu'il est possible de les arrêter avant la terminaison de l'algorithme et obtenir la solution

courante.

De fait, il existe de nombreux solveurs, qu'ils soient open-source comme *Gurobi* (MIP) ou *Choco-solver* (CP) ou non. Il a donc été nécessaire de faire un tri. Ce tri a été effectué en prenant en compte la popularité, la présomption d'efficacité des solveurs sur des problèmes d'ordonnancement ainsi que les méthodes variées, utilisées par les solveurs (MIP, CP, SAT, LS).

Voici la liste des solveurs qui vont être testés dans cette étude :

- *CPLEX Optimizer* (MIP)
- *CPLEX CP Optimizer* abbrégé CPO (CP)
- *CP-SAT Solver* (CP-SAT)
- *LocalSolver* (LS)

L'intérêt de ces solveurs est aussi de pouvoir comparer ceux déjà utilisé au laboratoire G-SCOP comme CPLEX et CPO à d'autres solveurs comme LocalSolver et CP-SAT.

CPLEX fournit des solveurs pour des modèles de *programmation linéaire*, *de programmation mixte en nombres entiers* (MIP) et de *programmation à contraintes quadratiques*. CPO est un solveur de CPLEX qui sert à résoudre des modèles de *programmation par contraintes* (CP). Ces solveurs sont basés sur la séparation et évaluation (*branch and bound*) ou par séparation et coupe (*branch and cut*). Récemment, la recherche dynamique et d'autres fonctionnalités ont été ajoutées pour permettre une résolution des modèles plus rapides. CPO contient un optimiseur qui gère les contraintes associées inhérentes à des problèmes de planification opérationnelle et d'ordonnancement selon Laborie et al. (2018). La recherche de CPO est elle aussi *anytime*.

CP-SAT est un solveur développé par Google, open-source et hybride puisqu'il intègre à la fois un moteur CP et un moteur SAT (Stuckey, 2012). Cette hybridation est appelée *génération paresseuse de clauses* (*Lazy Clause Generation*), Stuckey (2010) fourni une bonne explication du fonctionnement. Il s'est avéré performant pour des problèmes d'ordonnancement notamment le problème RCPSP (*Resource Constrained Project Scheduling Problems*) (Stuckey, 2012).

LocalSolver est un solveur d'optimisation qui possède son propre langage de modélisation et se base sur des heuristiques et, comme son nom l'indique la recherche locale (LS), ce qui le rend plus rapide et plus évolutif que les solveurs traditionnels (basés sur le *branch and bound*) en particulier pour l'optimisation de la chaîne logistique et les problèmes d'ordonnancement.

Les différents solveurs n'auront donc pas la même méthode de résolution pour ce problème d'ordonnancement. L'objectif est de voir quel solveur arrivera à résoudre ce problème le plus efficacement.

3 Modélisation du problème

3.1 *Mixed Integer Programming*

Deux modélisations MIP sont proposées, le but est de déterminer la modélisation la plus efficace. La différence entre les deux réside dans la modélisation de la **disjonction des tâches**, c'est-à-dire qu'une tâche ne peut pas être effectuée en même temps qu'une autre, seulement avant ou après. Ces deux modélisations seront implémentées sur le solveur *CPLEX Optimizer*.

Dans le premier modèle, les variables de décisions sont, pour tout i et j allant de 1 à n (n correspond au nombre de tâches) :

- y_{ij} : variable booléenne qui permet de modéliser la disjonction des tâches. Elle est égale à 1 si la tâche j est effectuée après la tâche i.
- C_j : variable qui correspond à la date de fin d'une tâche, nécessaire au calcul du retard.
- T_j : variable qui correspond au retard d'une tâche ($\max(0, C_j - d_j)$)

Voici le premier modèle :

$$\min w = \sum_{j=1}^{n} w_j T_j$$

$$\begin{aligned}
\text{s.c.} \quad & C_j \leq \tilde{d}_j && \forall\, j = 1, \ldots, n && (1)\\
& C_j \geq r_j + p_j && \forall\, j = 1, \ldots, n && (2)\\
& T_j \geq 0 && \forall\, j = 1, \ldots, n && (3)\\
& T_j \geq C_j - d_j && \forall\, j = 1, \ldots, n && (4)\\
& C_j \leq C_i - p_i + M y_{ij} && \forall\, i,\, j = 1, \ldots, n,\, j < i && (5)\\
& C_i \leq C_j - p_j + M(1 - y_{ij}) && \forall\, i,\, j = 1, \ldots, n,\, j < i && (6)\\
& y_{ij} \in \{0, 1\} && \forall\, i,\, j = 1, \ldots, n &&
\end{aligned}$$

Les contraintes (1) et (2) permettent le respect de la *deadline*, de la date d'arrivée et de la durée de la tâche j.

Les contraintes (3) et (4) permettent le calcul du retard T_j, présent dans la fonction objectif w.

Les contraintes (5) et (6) sont exclusives, une seule des deux peut être activée pour chaque tâche. Cela représente bien le fait qu'une tâche doit être effectuée avant ou après une autre. La constante M, nécessaire pour réaliser cette disjonction, est prise comme la *deadline* maximum de toutes les tâches puisque qu'une tâche ne peut pas commencer après sa *deadline*.

Pour notre autre modèle, la disjonction des tâches est cette fois-ci modélisée grâce au rang k de chaque tâche. Le rang k correspond à l'ordre d'exécution de la tâche j.

Les variables de décision du second modèle sont, pour tout k et j allant de 1 à n :

- a_{kj} : variable booléenne qui modélise l'attribution de la tâche j au rang k (égale à 1 si la tâche j est exécutée au rang k).
- C_k : variable qui correspond à la fin de la tâche au rang k
- T_k : variable qui représente le retard de la tâche au rang k ($\max(0, C_k - d_j a_{kj})$)
- T'_j : variable qui représente le retard de la tâche j

Voici le second modèle :

$$
\begin{aligned}
\min w = & \sum_{j=1}^{n} w_j T'_j & & \\
\text{s.c.} \quad & \sum_{j=1}^{n} a_{kj} = 1 & \forall\, k = 1, \ldots, n & \quad (1)\\
& \sum_{k=1}^{n} a_{kj} = 1 & \forall\, j = 1, \ldots, n & \quad (2)\\
& C_k \leq \sum_{j=1}^{n} \tilde{d}_j a_{kj} & \forall\, k = 1, \ldots, n & \quad (3)\\
& C_k \geq \sum_{j=1}^{n} (r_j + p_j) a_{kj} & \forall\, k = 1, \ldots, n & \quad (4)\\
& T_k \geq 0 & \forall\, k = 1, \ldots, n & \quad (5)\\
& T_k \geq C_k - \sum_{j=1}^{n} d_j a_{kj} & \forall\, k = 1, \ldots, n & \quad (6)\\
& C_{k-1} \leq C_k - \sum_{j=1}^{n} p_j a_{kj} & \forall\, k = 2, \ldots, n & \quad (7)\\
& M(1 - a_{kj}) + T'_j \geq T_k & \forall\, k,\, j = 1, \ldots, n & \quad (8)\\
& a_{kj} \in \{0, 1\} & \forall\, k,\, j = 1, \ldots, n &
\end{aligned}
$$

Les contraintes (1) et (2) assurent qu'une seule tâche j est exécutée au rang k et que toutes les tâches sont exécutées.

Les contraintes (3) et (4) permettent de respecter la *deadline*, la date d'arrivée et la durée de chaque tâche j.

Les contraintes (5) et (6) permettent de calculer le retard de la tâche au rang k.

La contrainte (7) assure que la tâche au rang k commence après la fin de la tâche au rang $k - 1$.

La contrainte (8) permet de calculer le retard de la tâche j à partir du retard calculé au rang k (T_k). Ce calcul est nécessaire pour éviter de multiplier deux variables de décision dans la fonction objectif w et nécessite la même constante M que celle du premier modèle *i.e.* la *deadline* maximum de toutes les tâches.

D'après Ku et Beck (2016), le premier modèle de disjonction serait plus efficace que le modèle du rang dû à Wagner (1959).

3.2 *Constraint Programming*

L'interêt principal de la *programmation par contraintes* en ordonnancement est la possibilité de définir des variables d'intervalle et d'associer des contraintes à ces variables. Une variable d'intervalle est une variable qui possède une date de début (*start*) et une date de fin (*end*) strictement supérieure à la date de début. Une telle variable possède donc une durée et est très utile pour modéliser une tâche. Les contraintes peuvent être, par exemple, le début au plus tôt d'une tâche, la fin au plus tard.

Voici le modèle CP (pour CPO) :

$$\begin{aligned}
&\min w = \sum_{j=1}^{n} w_j \max(endOf(I_j) - d_j, 0) \\
&\text{s.c.} \quad noOverlap(I) && && (1)\\
&\qquad sizeOf(I_j) = p_j && \forall j = 1, \ldots, n && (2)\\
&\qquad I_j \in [r_j, \tilde{d}_j] && \forall j = 1, \ldots, n && (3)
\end{aligned}$$

Les variables de décision sont les variables d'intervalle I_j pour tout j allant de 1 à n (n nombre de tâches). Dans la fonction objectif w, $endOf(I_j)$ permet de récupérer la date de fin choisie par le solveur pour la variable d'intervalle I_j.

La contrainte *noOverlap* (1) permet de faire la disjonction des tâches (I correspond à l'ensemble des intervalles I_j).

La contrainte (2) permet de dire que la variable d'intervalle I_j doit avoir une durée égale à la durée de la tâche j.

Enfin, la contrainte (3) est la définition des variables d'intervalle I_j avec comme contraintes le début minimum à la date d'arrivée r_j et la fin maximum à la *deadline* $\tilde{d}_j$.

Selon les solveurs, la mise en place du modèle CP est différente. Par exemple, dans CPO, il faut créer I_j de durée p_j puis mettre les contraintes $setStartMin(r_j)$ et $setEndMax(\tilde{d}_j)$. Il est aussi possible de récupérer la fin de tâche avec $endOf(I_j)$. Dans CP-SAT, il faut créer deux variables de décision intermédiaires qui correspondent aux dates de début et de fin choisies par le solveur. CPO est donc un peu plus intuitif que CP-SAT.

Voici ci-dessous, le code Python pour la modélisation CP-SAT, où les tâches sont stockées dans la variable *data* récupérée à partir d'une instance de la même forme que dans la section 2.1.

```
# Initialisation
model = cp_model.CpModel()
intervals = []
tardiness = []
```

```
# Pour chaque tache
for j in range(data['n']):
    # Variable correspondant au debut de la variable d'intervalle
    start = model.NewIntVar(
        data['r'][j],
        data["deadline"][j],
        name=f"start_job{j}")
    # Variable correspondant a la fin de la variable d'intervalle
    end = model.NewIntVar(
        data['r'][j],
        data["deadline"][j],
        name=f"end_job_{j}")
    # Variable d'intervalle de duree p
    intervals.append(model.NewIntervalVar(
        start,
        data['p'][j],
        end,
        name=f"job_{j}"))
    # Calcul du retard
    t = model.NewIntVar(0, data["deadline"][j], name=f"tardiness_{j}")
    model.AddMaxEquality(t, [end - data['d'][j], 0])
    tardiness.append(t)
# Contrainte pour la disjonction des taches
model.AddNoOverlap(intervals)
# Fonction objectif
model.Minimize(cp_model.LinearExpr.WeightedSum(tardiness, data['w']))
```

Il n'est pas inutile de remarquer qu'il est possible de réduire les intervalles *start* et *end*, contrairement au modèle CPO. En effet, dans le modèle CP-SAT présenté ci-dessus, nous n'avons pas fait de différence de domaine pour ces deux variables. Pourtant, la date de début ne pourra jamais être plus grande que la *deadline* moins la durée de la tâche et la date de fin ne pourra jamais être plus petite que la date d'arrivée pour la durée de la tâche. Réduire ces intervalles revient à diminuer l'espace de recherche. Malheureusement, ce détail n'a été remarqué qu'après les multiples expérimentations.

3.3 LocalSolver

La modélisation LocalSolver pour ce problème se base sur une liste de permutations (ordre des tâches). Elle peut s'apparenter à la logique du modèle MIP Rang car la date de fin d'une tâche est calculée à partir de la tâche précédente. Le solveur va effectuer des permutations et des calculs en fonction des contraintes données et va trouver une solution réalisable de bonne valeur, voire optimale.

Voici le code Python de la modélisation LocalSolver :

```
with localsolver.LocalSolver() as ls:
    model = ls.model
    # Liste des permutations
    job_order = model.list(data['n'])
    # Toutes les taches doivent etre realisees
    model.constraint(model.count(job_order) == data['n'])
```

```
# Calcul de la fin d'une tache
job_time_selector = model.lambda_function(lambda j, prev:
    model.iif(j > 0,
            model.max(prev, release_dates[job_order[j]])
            + processing_times[job_order[j]],
            release_dates[job_order[j]] + processing_times[job_order[j]]))
# Liste contenant la date de fin des taches dans l'ordre des permutations
C = model.array(model.range(0, data['n']), job_time_selector)
# Fonction objectif
objective = model.sum()
for i in range(data['n']):
    # Indice de permutation
    j = job_order[i]
    # Contraintes
    model.constraint(C[i] <= deadlines[j])
    model.constraint(C[i] >= release_dates[j] + processing_times[j])
    # Calcul du retard
    Tj = model.iif(C[i] > due_dates[j], C[i] - due_dates[j], 0)
    # Ajout a la fonction objectif
    objective.add_operand(Tj * weights[j])
model.minimize(objective)
model.close()
```

Pour le moment, le solveur n'arrive pas à déterminer si la solution trouvée est optimale ou non et la résolution s'arrête uniquement lorsque le solveur atteint la limite de temps imposée. Il trouve cependant les mêmes valeurs que les autres solveurs pour des instances données.

Ce modèle est celui qui donne les meilleurs résultats.

4 Comparaison et résultats des solveurs

4.1 Installation des solveurs

L'objectif de cette section est de garder une trace des méthodes d'installation pour les personnes souhaitant utiliser ces solveurs.

Les différentes installations sont décrites pour un système d'exploitation Linux et ont été testés sur la distribution Ubuntu. Pour les installations sur d'autres systèmes d'exploitation (OS - *Operating System*), il faut suivre les instructions sur les sites des solveurs respectifs.

Une API (*Application Protocol Interface*) est une interface informatique qui permet à des applications de communiquer entre elles et de s'échanger mutuellement des services ou des données. Ici, les API servent à communiquer avec les bibliothèques de code des solveurs pour des langages particuliers (Python, Java...).

Un IDE (*Integrated Development Environment*) est un logiciel qui contient des outils pour faciliter le développement logiciel.

4.1.1 OPL - CPLEX - CPO

Modeleur OPL

L'installation de l'IDE de *ILOG CPLEX Optimization Studio Community* (version bridée) n'est pas trop compliquée : il suffit de créer un compte IBM, de télécharger le dossier et suivre les instructions. A noter qu'il faut **2 Go** de libre pour l'installation. Sur Linux, après avoir téléchargé le fichier *.bin* il faut saisir dans un terminal, la commande suivante :

```
chmod u+x <nom_fichier>.bin
```

Ensuite, il ne reste plus qu'à le lancer et suivre l'installation dans le terminal. A la fin de l'installation, pour lancer l'IDE il faut ajouter le chemin :

```
<nom_dossier_installation>/opl/oplide
```

au *PATH*, par exemple dans le *.bashrc* et enfin taper la commande **oplide** dans le terminal.

Obtenir la version non bridée est un peu plus compliqué. Il faut créer un compte académique, aller sur la page pour obtenir la version académique, cliquer sur l'onglet *Software* puis ILOG CPLEX Optimization Studio, *Download*, choisir le bon OS, lire et accepter le contrat de licence et télécharger. Si ce n'est pas déjà le cas, il faut installer Java Web Start (**javaws icedtea-netx**) pour pouvoir lancer le téléchargement. La suite de l'installation se déroule comme précedemment.

API Python *docplex*

Il n'est pas nécessaire de télécharger l' IDE de IBM pour utiliser l'API Python bridée. En effet, il existe deux manières de procéder. Soit installer l'API avec l'utilitaire **pip** grâce à la commande suivante :

```
pip install docplex
```

Soit après installation de l'IDE (obligatoire pour la version non bridée), se déplacer dans le dossier suivant :

```
<nom_dossier_installation>/python
```

et exécuter la commande :

```
python setup.py install
```

Remarque. *Les commandes peuvent varier en fonction des systèmes Linux. Par exemple, parfois, il faut écrire* ***pip3*** *et* ***python3****.*

API Java

Il est nécessaire de télécharger l' IDE de IBM pour utiliser l'API Java. Ensuite, afin de faire fonctionner le projet Java, la première étape est d'importer le *.jar* du dossier :

```
<nom_dossier_installation>/opl/lib/oplall.jar
```

Pour configurer le lancement de l'application Java, il faut ajouter une variable d'environnement, par exemple dans le *.bashrc* :

```
export LD_LIBRARY_PATH=<nom_dossier_installation>/cplex/bin/x86-64_linux
       :$<nom_dossier_installation>/cpoptimizer/bin/x86-64_linux
```

Enfin pour que tout fonctionne, il faut ajouter dans les configurations de lancement du programme la ligne :

```
-Djava.library.path="<nom_dossier_installation>/opl/bin/x86-64_linux"
```

4.1.2 CP-SAT

API Python

L'installation pour Python (*Install OR-Tools*) est très simple et utilise l'utilitaire **pip**.

```
python -m pip install --upgrade --user ortools
```

API Java

L'installation pour Java est beaucoup plus compliquée. Il faut au préalable avoir une version du **Java JDK** ≥ 8, ainsi qu'une version de **Maven** ≥ 3.3. Après avoir téléchargé le binaire correspondant au bon OS, il faut le décompresser à l'endroit d'installation souhaité :

```
tar -xvf <or-tools...>.tar.gz
```

Après s'être déplacé dans le dossier décompressé, tester si l'installation fonctionne avec la commande :

```
make test_java
```

Pour utiliser la bibliothèque dans un nouveau projet Java, il faut récupérer ou créer un **pom.xml** avec les dépendances de la librairie et, compiler et lancer le projet avec Maven (**mvn compile run**). Il se trouve que la commande **make run SOURCE=<fichier>.java** seule comme indiqué sur le site ne fonctionne que pour les exemples fournis.

4.1.3 LocalSolver

L'installation de LocalSolver est très simple, il suffit de choisir son OS sur la page de téléchargement du site de LocalSolver et de suivre les instructions. Il faut cependant faire une demande de licence afin de pouvoir l'utiliser. Pour faire cette demande, il faut créer un compte, demander la licence acédémique et attendre qu'un membre de l'équipe LocalSolver valide ou non la demande de licence.

API Python

Après avoir installé LocalSolver, il faut, afin de pouvoir utiliser l'API Python, exécuter la commande suivante :

```
pip install localsolver -i https://pip.localsolver.com
```

4.2 Comparaison des solveurs

Dans cette section, nous allons comparer les solveurs sur deux critères, la documentation et la facilité d'utilisation. Cela permettra d'avoir un retour d'expérience et cela dans une perspective d'amélioration des solveurs.

4.2.1 OPL - CPLEX - CPO

Documentation

La documentation d'OPL est correcte mais celle de l'IDE pourrait être mieux. Il y a quelques exemples d'utilisation de l'API Python mais il est facile de se perdre dans le manuel de référence. En fait, il manque un menu de navigation. Ce qui est bien est la présence d'exemples d'utilisation à côté de la description des méthodes. Cela aide beaucoup pour comprendre comment s'en servir. Idem pour l'API Java. Il est assez difficile de naviguer de manière générale sur le site internet d'IBM car les temps de chargement des pages sont longs et les pages sont trop chargées, ou mal organisées.

Facilité d'utilisation

L'IDE pour OPL possède beaucoup de défauts. L'interface est plutôt agréable à utiliser mais le code n'est pas bien colorié à certains endroits (bug d'affichage) et son utilisation est vraiment limitée par rapport aux API. En revanche, le langage OPL est simple et plutôt facile à comprendre rapidement. Avec la logique de la programmation linéaire ou par contraintes ainsi que l'aide des IDE avec l'auto-complétion, les modèles CPLEX et CPO avec l'API Python sont relativement simples à implémenter. Le nom des méthodes à utiliser est assez clair et plusieurs possibilités de méthodes sont données à chaque fois, ce qui laisse un peu de liberté pour programmer.

4.2.2 CP-SAT

Documentation

Il manque des exemples pour l'utilisation du solveur pour des problèmes d'ordonnancement. Heureusement, les autres exemples fournis permettent de s'en sortir. Le manuel de référence est assez pauvre en informations, il y a des exemples mais les méthodes sont moins bien décrites comparé au manuel de CPLEX - CPO.

Facilité d'utilisation

Il faut vraiment bien comprendre la logique de la *programmation par contraintes* et certaines contraintes doivent être exprimées avec la logique SAT pour fonctionner. Ce mélange hybride est un peu perturbant au début et il est compliqué de trouver quelles variables doivent être exprimées grâce aux contraintes SAT ou non. Ce qui fait de CP-SAT un solveur CP moins facile à utiliser que CPO. De plus, la liberté de programmation semble assez limitée.

4.2.3 LocalSolver

Documentation

Les exemples d'ordonnancement pour l'API Python sont plutôt difficiles à comprendre au début parce que la modélisation avec LocalSolver est très différente des autres solveurs. Le fait de comprendre comment fonctionne le solveur aide vraiment pour modéliser son problème mais cela demande plus de temps. Le manuel de référence de l'API Python est correct même si des exemples d'utilisation à côté de certaines méthodes seraient utiles.

Facilité d'utilisation

Il faut bien comprendre comment LocalSolver cherche ses résultats pour pouvoir programmer un modèle correct. Les lambdas-fonctions sont un peu déroutantes au début mais cela permet vraiment une grande liberté de code. La mise en place du modèle est à peu près du même niveau de difficulté que CP-SAT.

4.3 Résultats des solveurs et interprétation

Les expérimentations ont toutes été réalisées sur une machine avec un processeur Intel Core i7-7500U CPU @ 2.70GHz × 4, une mémoire de 8 Go et un système d'exploitation Ubuntu 20.04.4 LTS, 64 bits.

Les expérimentations ont été réalisées avec les différentes API Python pour tous les solveurs et nous avons vérifié au préalable que tous les modèles sont corrects. Pour ce faire, nous avons vérifié sur des petites instances, que tous les solveurs retournent la même solution et grâce à des exemples jouets faits à la main.

Le code réalisé est disponible sur GitHub, dans le répertoire leapetitjean/solvers. Le code Python est divisé en plusieurs modules, un pour chaque solveur. Pour lancer des expérimentations, il faut lancer le fichier **main.py** avec un interpréteur Python.

Pour commencer, tous les solveurs (sauf LocalSolver parce qu'il ne s'arrête pas, même s'il trouve la solution optimale) ont été testés sur des instances générées selon la section 2.1 - Génération des instances. Chaque modèle est lancé sur son solveur respectif sur 30 instances différentes de même taille (nombre de tâches) par pas de 5 tâches. Le temps de résolution a été pris comme référence pour les petites instances.

La figure 1 montre les performances des solveurs avec une limite de temps, pour chaque solveur, fixée à 30 secondes et la moyenne du temps de résolution en fonction du nombre de tâches dans les instances.

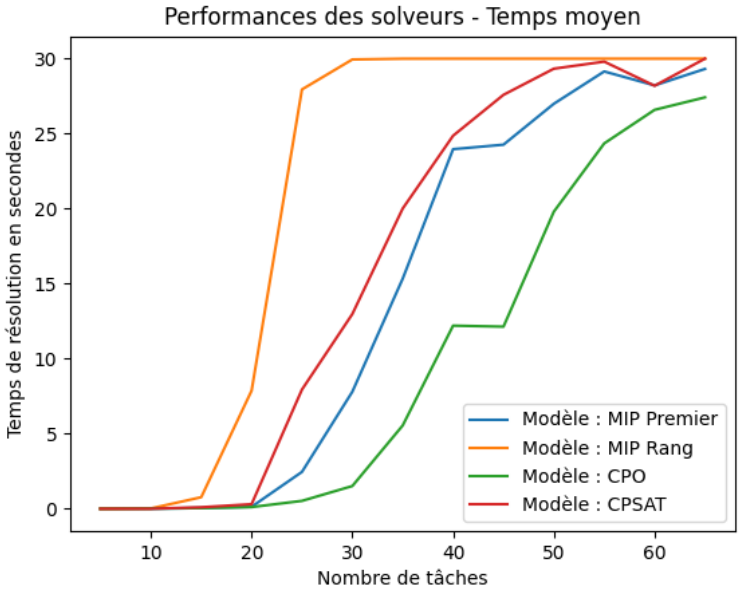


FIGURE 1 – Temps de résolution moyen des solveurs en fonction du nombre de tâches avec une limite de temps fixée à 30 secondes

La moyenne est un indicateur qui est sensible aux valeurs extrêmes, c'est pourquoi les résultats sont également présentés en figure 2 avec le temps médian de résolution (50%) ainsi

que le premier (25%) et dernier quartile (75%).

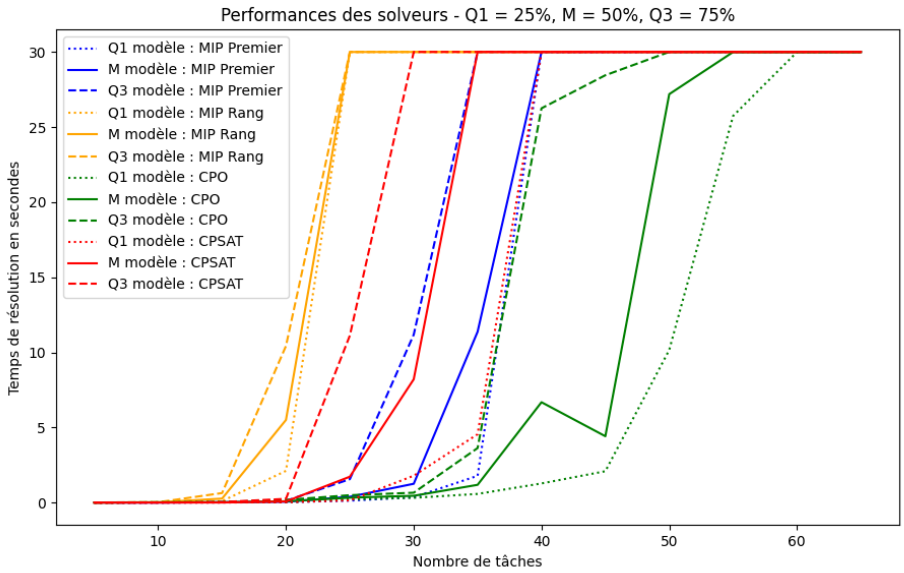


FIGURE 2 – Temps de résolution médian et au premier et dernier quartile des solveurs en fonction du nombre de tâches avec une limite de temps fixée à 30 secondes

Le modèle CPO semble le plus efficace pour les petites instances d'après les figures 1 et 2. Sans réelle surprise, comme annoncé dans le papier de KU et BECK (2016), le modèle de rang est beaucoup moins bon que les autres. D'après nos résultats, il est presque possible d'établir une hiérarchie des solveurs.

Cependant, en pratique, les industriels ont besoin de résoudre de plus grandes instances, c'est pourquoi il est nécessaire de faire d'autres comparaisons. Par ailleurs, lorsque la taille des instances grandit, il n'est plus possible de se baser sur les temps de résolution à l'optimal. D'une part parce que trouver l'optimal prendrait trop de temps et d'autre part parce qu'il faut inclure LocalSolver dans les expérimentations. La solution, inspirée de *Benchmark Archives*, est de définir un temps limite de résolution, de récupérer la valeur de la fonction objectif et de la comparer aux autres.

Les modèles MIP ne sont pas dans le tableau ci-dessous parce qu'à partir de 100 tâches, CPLEX ne renvoie plus de solutions.

Les colonnes 1^{er} et Écart des tableaux 1 et 2 correspondent respectivement au nombre de fois où le solveur a trouvé la meilleure solution S^* parmi les autres solveurs et à la moyenne des écarts entre la solution S trouvée par le solveur et la meilleure solution trouvée ($\frac{S-S^*}{S^*}$). La colonne Taille correspond au nombre de tâches des instances testées. Pour chaque taille d'instance (par pas de 100), 5 instances ont été lancées. Le nombre total d'instances par ligne est écrit à côté du nombre de fois où le solveur est arrivé premier, dans les colonnes Écart.

Le regroupement des lignes du tableau (par taille des instances) a été réalisé en fonction

de la proximité des résultats pour que la moyenne des écarts soit plus représentative. Par exemple, les instances de 100 à 900 tâches ont été regroupées parce que la majorité des résultats sont similaires. Enfin, la colonne Écart à l'optimal correspond à l'écart à l'optimal calculé par le solveur CPO.

Le paramètre d'inférence *noOverlapInference* est un paramètre qui concerne la propagation des contraintes de *noOverlap*. En fait, le terme inférence signifie admettre une proposition en raison de son lien avec une proposition préalable tenue pour vraie. Ce terme est notamment utilisé en intelligence artificielle symbolique, branche de l'intelligence artificielle qui repose sur la logique (Russell, Norvig et Popineau, 2010).

Le tableau 1 ci-dessous, nous permet de remarquer des changements selon les tailles des instances. Pour les plus petites instances de 100 à 900 tâches, LocalSolver trouve quasiment tout le temps la meilleure solution, suivi de très près par CPO. Ensuite, pour les instances de taille moyenne, CPO prend le dessus et au fur et à mesure que les instances grandissent LocalSolver s'éloigne de plus en plus de la meilleure solution alors que CP-SAT se rapproche de plus en plus de la meilleure solution. Enfin, sur les plus grandes instances, de 3000 à 3900 tâches, CP-SAT trouve les meilleures solutions, suivi encore une fois de très près par CPO alors que LocalSolver semble avoir beaucoup plus d'écart.

Taille	**CPO**			**CP-SAT**		**LocalSolver**	
	Écart à l'optimal	1^{er}	Écart	1^{er}	Écart	1^{er}	Écart
100-900	0.122	8/45	0.027	0/45	0.214	37/45	0.001
1000	0.191	4/5	0.002	0/5	0.192	1/5	0.024
1100-1200	0.165	10/10	0	0/10	0.141	0/10	0.166
1300-1700	0.200	25/25	0	0/25	0.111	0/25	0.573
1800-3400	0.203	82/85	0	3/85	0.101	0/85	1.094
3500-3900	0.295	0/25	0.033	25/25	0	0/25	4.311

Tableau 1 – Nombre de fois où les solveurs ont trouvé la meilleure solution (1^{er}) et écart entre la solution trouvée et la meilleure solution en fonction de la taille des instances avec une limite de temps fixée à 60 secondes et les paramètres de CPO sur *SearchType=Restart* et *noOverlapInference=Low*

Nous retrouvons les mêmes changements de dominance des solveurs en fonction de la taille des instances dans les résultats du tableau 2 ci-dessous qui a été obtenu avec les paramètres par défaut de CPO. Le type de recherche est par défaut en automatique (*Auto*) et l'inférence en *Basic*.

Remarquons que changer les paramètres de CPO change très légèrement les résultats, il semblerait que le paramètre d'inférence mis à *Low* (peu d'inférence) donne de moins bons résultats, en tout cas pour les plus petites instances (8 contre 15 et un écart à la meilleure solution plus important).

Nous pensons que ce changement de dominance est dû à plusieurs facteurs : la manière dont les solveurs gèrent les paramètres, certainement différemment selon la taille des instances, le fait que des heuristiques relatives à ce problème ont été implémentées ou non et la recherche de preuves d'optimalité qui prend certainement plus de temps, ce qui expliquerait pourquoi la recherche locale, sans preuve, donne de meilleures solutions au début.

Taille	CPO		CP-SAT		LocalSolver	
	1^{er}	Écart	1^{er}	Écart	1^{er}	Écart
100-900	15/45	0.009	0/45	0.217	30/45	0.004
1000	5/5	0	0/5	0.190	0/5	0.079
1100-1200	10/10	0	0/10	0.169	0/10	0.216
1300-1700	25/25	0	0/25	0.156	0/25	0.635
1800-3400	81/85	0.001	4/85	0.129	0/85	2.847
3500-3900	4/25	0.012	21/25	0.001	0/25	4.860

Tableau 2 – Nombre de fois où les solveurs ont trouvé la meilleure solution (1^{er}) et écart entre la solution trouvée et la meilleure solution en fonction de la taille des instances avec une limite de temps fixée à 60 secondes et les paramètres par défaut de CPO

Pendant les expérimentations, nous avons identifié des instances plus difficiles à résoudre que d'autres. En effet, certaines petites instances (de 40 tâches) arrivaient à être résolu avec une solution optimale en quelques secondes tandis que d'autres n'ont pas été résolu avec une solution optimale après plus de 15 minutes.

Conclusion

Bilan

Dans un premier temps, nous avons introduit l'ordonnancement et en particulier le problème, prouvé $\mathcal{NP}$-complet, $\mathbf{1} \,|\, \mathbf{r_j}, \tilde{\mathbf{d}}_\mathbf{j} \,|\, \sum \mathbf{w_j T_j}$. Nous avons aussi montré comment générer des instances de ce problème.

Dans un second temps, nous avons choisi et décrit le fonctionnement de quatre solveurs *CPLEX Optimizer*, *CPLEX CP Optimizer*, *CP-SAT Solver* et *LocalSolver*, puis donné et expliqué les modèles réalisés pour chaque solveur. Ensuite, nous avons explicité les différences d'installation, de documentation et de facilité d'utilisation des solveurs. Nous avons vu que toutes les installations pour l'API Python sont faciles, alors que celles pour Java sont beaucoup plus compliquées. Nous avons établi que le langage OPL et l'IDE de IBM sont très simples à installer et utiliser mais que les utilisations sont très limitées, c'est pourquoi l'utilisation d'API est plus intéressante.

Enfin, nous sommes arrivés à la partie résultats obtenus, où nous avons vu que tous les modèles (hors MIP), ont de très bons résultats. Sur les plus petites instances, LocalSolver semble être le plus efficace, même si, pour l'instant il ne peut pas prouver l'optimalité d'une solution au contraire de CPO et CP-SAT. De plus, il faut un peu de temps pour comprendre la logique du solveur et modéliser le problème, par rapport à CPO. En revanche, c'est le solveur le plus simple à installer. CP-SAT, quant à lui, est open-source et facile à installer. Il a le mérite d'avoir de très bons résultats quand la taille des instances est grande mais la modélisation et le fonctionnement du solveur sont assez difficiles à comprendre en plus d'être trop légèrement documenté. CPO semble être le solveur le plus adapté à ce problème d'ordonnancement et est pour moi, le plus simple à prendre en main malgré une installation un peu plus compliquée que les autres solveurs.

Perspectives

Dans cette section, nous présentons les pistes d'améliorations possibles pour cette analyse comparative des solveurs.

Premièrement, il serait intéressant d'étudier les instances qui sont plus difficiles à résoudre et pour cela, étudier leur structure et leurs caractéristiques.

Ensuite, il serait utile de changer la manière de générer les instances, en faisant varier la faisabilité et de comparer le temps où le solveur trouve la première solution réalisable. Nous pensons que la recherche locale aura plus de mal à trouver une solution réalisable que les méthodes qui utilisent la propagation de contraintes (CP).

Après coup, nous avons déjà remarqué qu'il est possible de réduire les domaines des variables *start* et *end* dans le code du solveur CP-SAT. Afin de voir s'il n'existe pas une meilleure façon de l'écrire, il faudrait prendre du temps pour mieux comprendre le fonctionnement du solveur et de voir s'il existe, comme le solveur CPO, des paramètres qui peuvent être changés pour améliorer ou non la recherche de solutions.

De plus, les expérimentations ont été effectuées avec un temps de résolution fixé à 60 secondes. Cependant pour les plus grandes instances, il serait pertinent de laisser plus de temps aux solveurs.

Enfin, il serait envisageable de vérifier ce qui rend le problème difficile à résoudre pour les solveurs, *i.e.* est-ce uniquement le calcul du retard ou la présence des poids des tâches ou encore trouver un ordonnancement réalisable qui donne du fil à retordre aux solveurs ? Nous sommes d'avis que la présence des poids complique radicalement la recherche de solutions optimale.

Et, éventuellement, ajouter d'autres solveurs à cette étude comparative.

Références

Bessière, Christian et al. (2005). « An optimal coarse-grained arc consistency algorithm ». In : *Artificial Intelligence* 165.2, p. 165-185. ISSN : 0004-3702. DOI : https://doi.org/10.1016/j.artint.2005.02.004. URL : https://www.sciencedirect.com/science/article/pii/S0004370205000482.

Chekuri, Chandra et Rajeev Motwani (1999). « Precedence constrained scheduling to minimize sum of weighted completion times on a single machine ». In : *Discrete Applied Mathematics* 98.1, p. 29-38.

Cheng, Runwei, Mitsuo Gen et Yasuhiro Tsujimura (1996). « A tutorial survey of job-shop scheduling problems using genetic algorithms—I. representation ». In : *Computers Industrial Engineering* 30.4, p. 983-997. ISSN : 0360-8352. DOI : https://doi.org/10.1016/0360-8352(96)00047-2. URL : https://www.sciencedirect.com/science/article/pii/0360835296000472.

Choco-solver. *Choco-solver*. URL : https://choco-solver.org/.

Cloud team, The IBM Decision Optimization on. *docplex*. URL : https://pypi.org/project/docplex/.

« Definition, Analysis and Classification of Scheduling Problems » (2007). In : *Handbook on Scheduling : From Theory to Applications*. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 57-72. ISBN : 978-3-540-32220-7. DOI : 10.1007/978-3-540-32220-7_3. URL : https://doi.org/10.1007/978-3-540-32220-7_3.

Garey, M. R. et David S. Johnson (1978). « Computers and Intractability : A Guide to the Theory of NP-Completeness ». In.

Gotha (1993). « Les problèmes d'ordonnancement ». fr. In : *RAIRO - Operations Research - Recherche Opérationnelle* 27.1, p. 77-150. URL : http://www.numdam.org/item/RO_1993__27_1_77_0/.

Hoogeveen, Han, Petra Schuurman et Gerhard J Woeginger (2001). « Non-approximability results for scheduling problems with minsum criteria ». In : *INFORMS Journal on Computing* 13.2, p. 157-168.

IBM. *CPLEX CP Optimizer*. URL : https://www.ibm.com/fr-fr/analytics/cplex-cp-optimizer.

— *CPLEX Optimizer*. URL : https://www.ibm.com/fr-fr/analytics/cplex-optimizer.

— *ILOG CPLEX Optimization Studio*. URL : https://www.ibm.com/products/ilog-cplex-optimization-studio.

KING, J. R. et A. S. SPACHIS (1980). « Heuristics for flow-shop scheduling ». In : *International Journal of Production Research* 18.3, p. 345-357. DOI : 10.1080/00207548008919673. eprint : https://doi.org/10.1080/00207548008919673. URL : https://doi.org/10.1080/00207548008919673.

Ku, Wen-Yang et J. Beck (avr. 2016). « Mixed Integer Programming Models for Job Shop Scheduling : A Computational Analysis ». In : *Computers & Operations Research* 73. DOI : 10.1016/j.cor.2016.04.006.

Laborie, Philippe et al. (avr. 2018). « IBM ILOG CP optimizer for scheduling ». In : *Constraints* 23.2, p. 210-250. ISSN : 1572-9354. DOI : 10.1007/s10601-018-9281-x. URL : https://doi.org/10.1007/s10601-018-9281-x.

Lawler, E.L. (1977). « A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness ». In : *Ann. of Discrete Math.* 1, p. 331-342.

Lawler, Eugene L et J Michael Moore (1969). « A functional equation and its application to resource allocation and sequencing problems ». In : *Management science* 16.1, p. 77-84.

Leung, J.Y.T. (2004). *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis. isbn : 9781135438852. url : https://books.google.fr/books?id=0UV5AgAAQBAJ.

LocalSolver. *Benchmark Archives*. url : https://www.localsolver.com/benchmark.

— *LocalSolver*. url : https://www.localsolver.com/product.html.

Malik, Sharad et Zhang Lintao (août 2009). « Boolean satisfiability from Theoretical hardness to Practical success ». In : *Commun. ACM* 52, p. 76-82. doi : 10.1145/1536616.1536637.

Müller-Merbach, Heiner (1981). « Heuristics and their design : a survey ». In : *European Journal of Operational Research* 8, p. 1-23.

Optimization, Gurobi. *Gurobi*. url : https://www.gurobi.com/.

Pinedo, Michael L. (2012). « Constraint Programming ». In : *Scheduling : Theory, Algorithms, and Systems*. Boston, MA : Springer US, p. 581-588. isbn : 978-1-4614-2361-4. doi : 10.1007/978-1-4614-2361-4_23. url : https://doi.org/10.1007/978-1-4614-2361-4_23.

Roselli, Sabino, Kristofer Bengtsson et Knut Åkesson (août 2018). « SMT Solvers for Job-Shop Scheduling Problems : Models Comparison and Performance Evaluation ». In : p. 547-552. doi : 10.1109/COASE.2018.8560344.

Ruiz, Rubén et Marco Pranzo (juill. 2006). « Holger H. Hoos, Thomas StützleStochastic Local Search : Foundations and Applications2005, Morgan Kaufmann Publishers ». In : *European Journal of Operational Research* 172, p. 716-718. doi : 10.1016/j.ejor.2005.04.002.

Russell, S., P. Norvig et F. Popineau (2010). *Intelligence artificielle : Avec plus de 500 exercices*. Pearson Education. Pearson. isbn : 9782744074554. url : https://books.google.fr/books?id=DWTlFWSGxJMC.

Skutella, Martin (2006). « List scheduling in order of α-points on a single machine ». In : *Efficient Approximation and Online Algorithms*. Springer, p. 250-291.

Stuckey, Peter J. (2010). « Lazy Clause Generation : Combining the Power of SAT and CP (and MIP ?) Solving ». In : *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Sous la dir. d'Andrea Lodi, Michela Milano et Paolo Toth. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 5-9. isbn : 978-3-642-13520-0.

— (2012). url : https://people.eng.unimelb.edu.au/pstuckey/PPDP2013.pdf.

OR-Tools, Google. *CP-SAT Solver*. url : https://developers.google.com/optimization/cp/cp_solver.

— *Install OR-Tools*. url : https://developers.google.com/optimization/install.

Wagner, Harvey M. (1959). « An integer linear-programming model for machine scheduling ». In : *Naval Research Logistics Quarterly* 6, p. 131-140.