Ordonnancement disjonctif en variables d'intervalles avec Hexaly Optimizer

Léa Blaise[1]

Hexaly, France
lblaise@hexaly.com

Mots-clés : *ordonnancement, solveur, modélisation, recherche locale, propagation*

1 Introduction

Hexaly Optimizer, anciennement LocalSolver, est un solveur global de type *model & run* basé sur des techniques exactes et heuristiques [2]. On s'intéresse ici à ses performances sur les problèmes d'ordonnancement disjonctif de façon générale (problèmes présentant des contraintes de précédence et non-chevauchement des tâches, minimisant le makespan, la somme pondérée des retards, etc.). On montrera d'une part comment son formalisme de modélisation permet d'exprimer de nombreux problèmes d'ordonnancement disjonctif académiques et industriels, en n'utilisant que des opérateurs génériques. Ces modèles, construits à partir de variables d'intervalles et de listes, ont l'avantage d'être compacts, et permettent ainsi au solveur de traiter des problèmes de grande taille.

On montrera également comment le solveur exploite cette modélisation pour fournir des solutions de qualité rapidement, à travers des mouvements de recherche locale et un algorithme de réparation de solutions par propagation. Hexaly Optimizer se distingue ainsi des autres solveurs classiques pour l'ordonnancement, comme CP Optimizer ou OR-Tools, qui exploitent principalement des techniques de programmation par contraintes. Les algorithmes implémentés au sein d'Hexaly Optimizer lui permettent d'obtenir de très bonnes performances sur divers problèmes d'ordonnancement disjonctif : 2.2% d'écart à la meilleure solution connue sur le Job Shop jusqu'à 2000 tâches, 0.3% sur le Flexible Job Shop jusqu'à 500 tâches, 0.1% sur l'Open Shop jusqu'à 400 tâches, le tout après une minute de calcul.

2 Modélisation des problèmes d'ordonnancement avec des variables d'intervalles et de listes

La présence de ressources disjonctives dans un problème d'ordonnancement se caractérise par des contraintes de non-chevauchement des tâches affectées à une même ressource. Une façon simple d'écrire cette contrainte consiste à exploiter l'ordre des tâches : chaque tâche ne peut commencer qu'après la fin de la tâche précédente.

L'écriture de cette contrainte dans le formalisme de modélisation d'Hexaly Optimizer se fait en utilisant deux types de variables de décision. D'une part, des variables d'intervalles permettent de représenter les plages d'exécution des tâches. D'autre part, chaque ressource est représentée à l'aide d'une variable de liste [1], correspondant à l'ensemble des tâches exécutées sur cette ressource, rangées par dates d'exécution croissantes.

```
task[0...nbTasks] <- interval(0, horizon);
order <- list(nbTasks);
```

1. Variable de décision dont la valeur est une permutation d'un sous-ensemble de $\{0, ..., n-1\}$.

On écrit alors la contrainte de non-chevauchement à l'aide d'un opérateur « et » variadique. Cette contrainte se lit « pour toute position i dans la liste, la tâche en position i doit être exécutée avant la tâche en position $i+1$ ». Cette formulation à base de variables de listes et d'intervalles a l'avantage de permettre à l'utilisateur d'écrire la contrainte de non-chevauchement en $O(n)$ seulement, n représentant le nombre de tâches, contre $O(n^2)$ en considérant les tâches deux à deux et en n'utilisant que des variables d'intervalles.

```
constraint and(0...nbTasks-1, i => task[order[i]] < task[order[i+1]]);                    // (1)
```

Ces contraintes sont à la base de nombreux problèmes d'ordonnancement disjonctif, comme par exemple celui du Job Shop.

3 Recherche locale à voisinages restreints

En plus d'offrir une modélisation pratique des contraintes de non-chevauchement, l'expression (1) permet de donner une structure forte au modèle, exploitable par le solveur, notamment dans les mouvements de la recherche locale.

Différents mouvements propres à l'ordonnancement sont ainsi implémentés au sein de la composante de recherche locale d'Hexaly Optimizer, automatiquement activés ou désactivés en fonction des caractéristiques des ressources et des tâches (durées fixes ou variables notamment). On peut citer les exemples suivants : insertion d'une nouvelle tâche sur une machine, fusion de deux tâches voisines, changement de ressource, échange de ressource entre deux tâches, échange des dates de début de k tâches affectées à une même ressource, fractionnement d'une tâche, inversion de l'état de fonctionnement de k ressources sur une petite fenêtre de temps...

Ces mouvements exploitent bien l'interaction entre les variables représentant les plages d'exécution des tâches et leur ordre sur les machines. Par exemple, pour le mouvement de changement de ressource, on retire l'indice de la tâche choisie de la liste de sa machine initiale pour l'ajouter à la liste d'une autre machine à la bonne position, s'assurant ainsi que les tâches restent triées par dates de début croissantes.

4 Réparation de solutions par propagation de contraintes

Seule, cette recherche locale à voisinages restreints ne permet cependant pas au solveur d'obtenir de bonnes performances sur les problèmes d'ordonnancement disjonctif. En effet, dans une bonne solution d'un tel problème, les contraintes de précédence et de non-chevauchement entre les tâches sont très serrées. Le passage d'une bonne solution à une solution légèrement meilleure nécessite alors de modifier les plages d'exécution d'un grand nombre de tâches.

Afin de palier ce problème, un algorithme de réparation de solutions par propagation de contraintes [1] a été implémenté au sein d'Hexaly Optimizer. Le principe de cet algorithme est le suivant. Lorsque la solution obtenue à l'issue d'un mouvement de recherche locale est infaisable, celle-ci est graduellement réparée, une contrainte à la fois. Le but de cet algorithme est d'obtenir une solution faisable en étendant un mouvement de recherche locale prometteur mais infaisable, plutôt qu'en l'annulant. Ainsi, on impose de réparer les contraintes en modifiant les valeurs des variables toujours dans le même sens. Par exemple, si une tâche a déjà été reculée, lors du mouvement ou d'une précédente réparation, celle-ci pourra être reculée davantage lors d'une réparation future, mais ne pourra pas être avancée.

Références

[1] L. Blaise, C. Artigues, and T. Benoist. Solution Repair by Inequality Network Propagation in LocalSolver. In *Parallel Problem Solving from Nature – PPSN XVI*, 2020.

[2] F. Gardi, T. Benoist, J. Darlay, B. Estellon, and R. Megel. *Mathematical Programming Solver Based on Local Search.* John Wiley & Sons, Ltd, 2014.