Résolution du problème de l'Assembly Line Balancing avec LocalSolver

Léa Blaise[1,2], Thierry Benoist[2], Christian Artigues[1]

[1] LAAS-CNRS, Université de Toulouse, CNRS, INP, Toulouse, France
[2] LocalSolver, 24 Avenue Hoche, Paris, France
lblaise@localsolver.com

Mots-clés : *recherche locale, ordonnancement, packing, solveur.*

1 Introduction

LocalSolver est un solveur d'optimisation mathématique basé sur différentes techniques de recherche opérationnelle, combinant des méthodes exactes, telles que la programmation linéaire, non linéaire et par contraintes, et heuristiques, comme la recherche locale[1]. Son but est d'offrir une approche de type "model-and-run" à des problèmes d'optimisation (combinatoires, continus, mixtes...), y compris sur de grandes instances.

On s'intéresse ici au problème de l'Assembly Line Balancing, et on montre comment les algorithmes implémentés au sein de LocalSolver permettent d'obtenir d'excellents résultats sur ce problème. Le problème de l'Assembly Line Balancing est décrit de la façon suivante. On considère un ensemble de n tâches, de durées fixes, et partiellement ordonnées par des relations de précédence. Le problème consiste à répartir les tâches dans des « stations de travail », en respectant les relations de précédence entre les tâches, et en s'assurant que la somme des durées des tâches présentes dans une même station ne dépasse pas le temps de cycle c. L'objectif est de minimiser le nombre de stations utilisées.

La modélisation du problème avec LocalSolver se fait en utilisant des variables ensemblistes : chaque station de travail S est modélisée par une variable de set `stations[s]`, dont la valeur est égale à l'ensemble des tâches qu'elle contient. Chaque tâche devant être affectée à exactement une station, on contraint l'ensemble des variables de sets à former une partition.

2 Algorithme constructif

La contrainte de respect de la relation d'ordre entre deux tâches `t1` et `t2` s'écrit :

```
constraint find(stations, t1) <= find(stations, t2);
```

Lorsque l'on détecte ce type de contraintes de précédence dans le modèle, on en déduit une relation d'ordre sur les variables de sets du modèle. On peut alors élaborer un algorithme constructif pour un mouvement de recherche locale de type « destroy and repair », réorganisant les éléments présents dans une partie des variables de sets du problème, en tenant compte des contraintes de précédence entre les tâches, et des contraintes de capacité sur les stations de travail. La structure de l'algorithme est la suivante. On sélectionne un ensemble de variables de sets consécutives, rangées par ordre croissant, que l'on vide de leurs éléments. On réassigne ensuite ces éléments dans les différents sets sélectionnés. On commence par remplir la première variable de set S. Tant qu'il existe un élément t sans prédécesseur non assigné, et de poids suffisamment faible pour être ajouté à S, on ajoute t à S. Sinon, on passe à la variable de set suivante, et ainsi de suite. Appliqué à l'ensemble des variables de sets du modèle, cet algorithme permet également de construire très vite une première solution initiale réalisable au problème. Dans les deux cas, la complexité de l'algorithme est de $O(N^2)$, où N est le nombre total d'éléments considérés.

L'implémentation de cet algorithme étant la plus générique possible, il ne s'applique pas seulement au problème de l'Assembly Line Balancing, mais à n'importe quel problème présentant une structure d'ordre et une structure de packing sur ses variables de sets.

3 Mouvement de packing à base de chaînes d'éjection

On décrit ici un autre mouvement de la recherche locale de LocalSolver, exploitant la structure de packing détectée dans le modèle (somme des durées des tâches d'une station inférieure au temps de cycle). Ce mouvement est basé sur le principe des chaînes d'éjection : il consiste en une série de déplacements d'éléments d'une variable de set vers une autre.

On commence par choisir un sous-ensemble de variables de sets du modèle. On note S la variable de set ayant le plus petit poids parmi ceux-ci. On éjecte un élément t choisi aléatoirement dans S. Le but du mouvement est de réorganiser les éléments présents dans les autres variables de sets sélectionnées, afin de pouvoir y insérer t. Pour cela, on sélectionne le plus petit élément t' pouvant être remplacé par t. On l'éjecte de sa variable de set S', et on insère t à sa place. On recommence alors la même procédure, avec t' comme élément courant, jusqu'à pouvoir l'insérer dans une nouvelle variable de set sans avoir à réaliser de nouvelle éjection (succès), ou jusqu'à ce qu'il n'existe plus de nouvelle éjection possible (échec). La complexité du mouvement est de $O(N^2)$, avec N le nombre total d'éléments présents dans les sets choisis.

Ce mouvement est particulièrement efficace sur les instances très combinatoires, dans lesquelles chaque variable de set ne contient que très peu d'éléments : il est alors souvent améliorant en cas de succès. Il est cependant également utile sur d'autres types d'instances : en creusant les écarts de poids entre les différentes variables de sets, il permet de trouver des améliorations plus facilement dans les itérations suivantes. Ce mouvement s'applique sur tous les problèmes présentant une structure de packing. Il améliore ainsi les performances de LocalSolver sur différents problèmes, tels que ceux du Bin Packing et de l'Assembly Line Balancing.

4 Résultats

On compare les performances des solveurs LocalSolver 10.5, CP Optimizer 20.1.0, et Gurobi 9.1, sur le problème de l'Assembly Line Balancing, en 60 et 600 secondes de calcul. On utilise pour cela un benchmark de 525 instances de 1000 tâches (instances « *very large* » proposées par Otto *et al.* dans [2]). La métrique utilisée est l'écart à la meilleure solution connue, égale au minimum entre la meilleure solution connue présentée dans [2] et la valeur obtenue par LocalSolver au bout de 10 minutes de calcul. En effet, LocalSolver améliore le résultat donné par les auteurs de l'article sur 59% des instances. Les modèles utilisés pour évaluer les performances de CP Optimizer et Gurobi sont respectivement donnés dans [3] et [4].

	LocalSolver		CP Optimizer		Gurobi	
	60s	600s	60s	600s	60s	600s
Instances faisables	100%	100%	100%	100%	0%	0%
Instances gap $< 1\%$	95%	99%	59%	64%	0%	0%
Gap moyen	0.4%	0.1%	2.1%	1.7%	–	–

TAB. 1 – Comparaison des performances de LocalSolver, CP Optimizer et Gurobi

Les performances de LocalSolver sont nettement supérieures à celles des deux autres solveurs, avec un écart très significatif sur les instances les plus combinatoires, réputées les plus difficiles.

Références

[1] Frédéric Gardi, Thierry Benoist, Julien Darlay, Bertrand Estellon, and Romain Megel. *Mathematical Programming Solver Based on Local Search*, Wiley, 2014.

[2] Alena Otto and Christian Otto and Armin Scholl. Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing. *European Journal of Operational Research*, 2013.

[3] Laborie, Philippe. Solving the Simple Assembly Line Balancing Problem with CP Optimizer. https ://www.linkedin.com/pulse/solving-simple-assembly-line-balancing-problem-cp-philippe-laborie/, 2020.

[4] Pastor, Rafael and Ferrer, Laia and García, Alberto. Evaluating optimization models to solve SALBP. Lecture Notes in Computer Science, 2007.