POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

INF8215
TP3 - Ordonnancement de la construction de voitures

Nom d'équipe: IA
Auteurs:
Lamia Salhi 2164386
Theo Delbecq 2161923

Professeur:
Quentin Cappart
Gaël Reynal

2021-2022

Contents

1 Introduction

Nous souhaitons optimiser la production simultanée de voitures sur les infrastructures d'une usine. La construction de chaque voiture est une tâche (*job*) composée de plusieurs opérations qui peuvent être réalisées sur divers machines avec une certaine durée. Une solution est un planning détaillant les opérations et les temps d'exécutions sur chaque machine. Deux contraintes dures majeures s'imposent : les opérations d'une tâche doivent être réaliser dans l'ordre strictement et une machine ne peut réaliser qu'une opération à la fois. L'objectif est de minimiser le temps nécessaire pour compléter l'ensemble des opérations (*makespan*);
Ce problème est une variante plus complexe du Job Shop Scheduling Problem (JSSP). C'est un problème NP-difficile nous avons donc essayé de le résoudre de manière approchée en testant plusieurs métaheuristiques.

2 Présentation de la solution

2.1 Espace de représentation du problème

Le planning est représenté par un chromosome avec $n_oprations$ (le nombre d'opération total) gènes, tel que défini dans les papiers [1] et [2]. Un planning correspond alors en une liste d'entiers représentants les opérations. Chaque opération est représenté par l'entier de la tâche à laquelle elle appartient. Donc chaque job/tâche j est représenté $n_{oprations_j}$ (le nombre d'opérations du job j). Par exemple, pour un chromosome [1 2 2 3 1 2 3 3] où 1, 2, 3 représente les tâches j_1, j_2, j_3 respectivement. Ici, j_1 possède 2 opérations et j_2 et j_3 possèdent 3 opérations. Le chromosome [1 2 2 3 1 2 3 3] peut être lu, [$o_{1,1}$ $o_{2,1}$ $o_{2,2}$ $o_{3,1}$ $o_{1,2}$ $o_{2,3}$ $o_{3,2}$ $o_{3,3}$] où $o_{j,k}$ correspond à la k ième opération de la tache j [1],[2].
Pour décoder le chromosome et donc revenir à une représentation (machine, job, opération, début de l'opération, fin de l'opération) on utilise une heuristique d'interprétation déterministe afin d'avoir une solution complète et une valeur d'évaluation fixe pour chaque chromose. Nous avons testé deux décodages:
- La première consiste à assigner, itérativement, l'opération suivante des jobs dans l'ordre donné par le chromosome. Chaque assignation se fait à la machine qui peut réaliser l'opération en induisant le plus faible makespan. L'ajout de l'opération se fait en bout de file. On l'appelle le décodage de base.
- La seconde méthode ajoute à la première la possibilité d'insérer une opération dans un trou laissé par les précédentes assignations sur le fil d'exécution d'une machine tout en respectant la précédence et le non-chevauchement. En cas d'égalité, c'est la 1ère assignation avec le temps de départ le plus faible qui est choisie. On l'appelle le décodage greedy.
Diverse variantes d'implémentations de ces méthodes ont été testées en faisant varier la priorité d'assignation, les règles de tie ou les méthodes de calcul. En effet, le décodage est utilisé pour l'évaluation et, est donc utilisé très souvent, donc c'est une opération chère en temps et fortement impactante sur la qualité de la solution. Les meilleures implémentations sont comparées dans la section 3.
Finalement, cette façon de gérer l'espace permet de valider en tout temps les contraintes dures au moment du décodage tout en nous permettant d'avoir des voisinages connectés. Il est nécessaire que ces opérations soient déterministes car avec ces méthodes de décodage on n'a pas la bijectivité entre l'espace des chromosomes et celui des solutions. Cependant on a la surjectivité, notamment dans le sens des chromosomes vers les solution qui garantit la possibilité d'accéder à une potentielle solution optimale.

2.2 Voisinages considérés

Nous avons considéré 3 voisinages : - Le **2-swap** mais seulement entre 2 opérations i et i+1 côte à côte. Par exemple, pour i = 2 le chromosome voisin de [1 2 2 3 1 2 3 3] est [1 2 3 2 1 2 3 3].
- L' **interchange** qui prend 2 opérations i et j et les échange. Par exemple, pour i = 2 et j=6 le chromosome voisin de [1 2 2 3 1 2 3 3] est [1 2 3 3 1 2 2 3].
- l' **insert** qui prend 2 opération i et j et déplace l'opération i juste avant l'opération j. Par exemple, pour i=2 et j=7 le chromosome voisin de [1 2 2 3 1 2 3 3] est [1 2 3 1 2 3 2 3].

2.3 Métaheuristique de recherche

La méthode choisie est une General Variable Neighbor Search (GVNS) avec les points de conceptions suivants:
- **Initialisation :** On retient la meilleure solution aléatoire sur 100 générés uniformément (intéressant jusqu'à 1000 sur les grandes instances). Une initialisation par colonie de fourmis a été testée avec une heuristique gloutonne. Celle-ci s'est révélée moins bonne sur les grandes instances.
- **Validité :** L'ensemble des voisins est valide car l'espace permet de vérifier les contraintes dures en tout temps.
- **Sélection :** On sélectionne le meilleure voisin à chaque itération. On a testé la sélection du premier voisin améliorant qui n'est pas rentable. (On fait plus d'itérations sur le temps imparti mais on atteint jamais le même niveau de convergence)
- **Évaluation :** L'évaluation se fait en décodant le chromosome et en retournant le temps de fin maximal des opérations. La qualité dépend beaucoup du décodeur choisi : basique ou greedy.
- **Intensification :** D'abord une recherche à voisinage variable (VND) a été implémentée. Le choix du meilleur voisin été remplacé par une VND elle-même pour mettre en place la GVNS qui est strictement améliorante.
- **Diversification :** Pour mitiger l'intensification induite par la GVNS et la stagnation notamment sur les plus grandes instances deux mécanismes de diversification ont été ajoutés en montrant des améliorations. On ajoute d'abord des restarts après 10 itérations sans amélioration (autres paramètres testés : $5 * nb_{jobs}, 10 * nb_{jobs}, nb_{totaloperations}$) ce qui permet de compenser les voisinages à petit mouvements. Une fonction de shake pour induire de l'aléatoire contrôlé avant la VND interne sur le modèle de la BVNS afin d'introduire un mouvement plus important sans être obligé de restart.

Enfin, divers ensembles et ordre de considération des voisinages ont été testés afin de déterminer la meilleure façon de prioriser les types de mouvements (cf. section 3).
Voici le pseudo code pris du papier [3] :

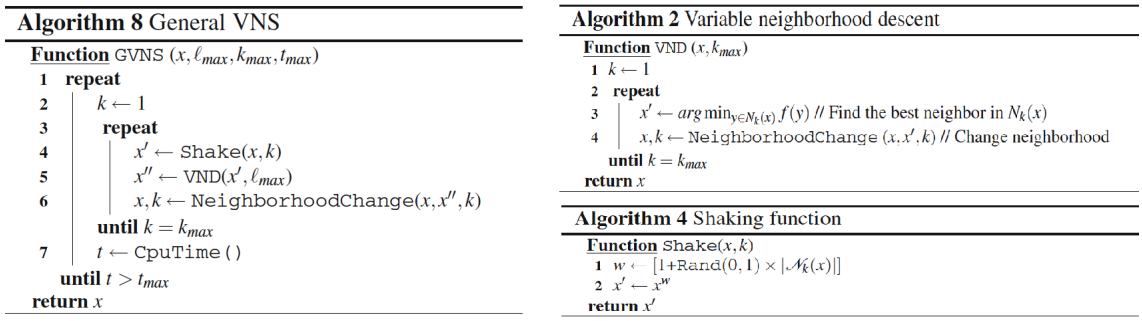

Figure 1: Pseudo code GVNS [3]

3 Résultats et discussions

3.1 Résultats

Nous avons testé plusieurs GVNS qui diffèrent sur l'agencement des voisinages. Pour cela nous avons utilisé le décodeur de base (assignation d'une opération à la machine de plus faible makespan). Voici les configurations :
V.0 : 2-swap –> interchange **sans** shake
V.1 : 2-swap –> interchange avec shake
V.2 : interchange –> 2-swap avec shake
V.3 : 2-swap -> interchange –> insertion avec shake
V.4 : interchange –> 2-swap –> insertion avec shake

NB : Les instances D et E sont les résultats après 20 minutes. Les algorithmes n'ont pas eu le temps de finir leur descente.

Résultats GVNS avec décodeur de base par instance					
	instance A	instance B	instance C	instance D	instance E
naive search	86	853	1479	1892	2416
V.0	47	514	756	917	1036
V.1	47	504	756	822	1012
V.2	47	506	756	823	971
V.3	47	504	756	836	1010
V.4	47	512	756	846	994
optimum	47	502	756	inconnu	inconnu

Les principales stratégies de décodages utilisées sont les suivantes : le décodage de base, le décodage greedy (voir section 2.1) et le décodage hybride qui consiste en l'utilisation du décodage de base sauf pour la dernière solution rendue qui est décodée avec la méthode greedy si cela donne de meilleurs résultats.

Résultats des méthodes de décodage					
	instance A	instance B	instance C	instance D	instance E
Décodeur basique	47	504	756	829	989
Décodeur greedy	47	507	756	829	993
Décodeur hybride	47	505	756	828	980
optimum	47	502	756	inconnu	inconnu

3.2 Discussion

Tout d'abord, dans le premier tableau on peut voir l'intérêt du "shake" qui nous permet de drastiquement améliorer nos résultats entre le GVNS V.0 et le V.1. Ensuite, on peut voir que l'ordre dans lequel les voisinages sont traités influent sur nos résultats. Commencer par des petits voisinages (swap) puis par des plus gros voisinages (interchange) permet une meilleur convergence en général car le pas de convergence du premier voisin est petit. Cependant, la convergence est plus lente qu'en commençant par de plus gros voisinages. On peut le voir ici, V.1 donne de meilleurs résultats que V.2 sur l'instance B. Le pas de convergence de V.2 est alors trop grand pour converger plus. Cependant, lorsque les instances sont plus grandes et que l'algorithme n'a pas le temps de converger complètement (instances D et E), c'est la V.2 qui donne de meilleurs résultats car la convergence est plus rapide. Les mêmes résultats sont visibles sur V.3 et V.4. Enfin, ajouter l'insertion n'améliore pas nos résultats sur les petites instances, et donne de moins bon résultats sur les grosses instances. En effet, ajouter un voisinages demande plus de temps de calculs et donc peut ralentir la convergence. Ce qui est notre cas ici. Il y a donc un équilibre à trouver entre performance et temps d'exécution.

Concernant le décodage, en général, le décodeur greedy permet une évaluation égale ou meilleure mais le coût est quadratique (division par 3 à 8 du nombre d'itérations selon la taille de l'instance). Ce coût est rédhibitoire pour les grandes instances où le nombre d'itération devient très faible et donc peu robuste. De plus, la méthode greedy semble moins bonne sur l'instance B notamment. Ses critères de constructions plus restrictifs doivent limiter les possibilités de mouvement et donc même si on améliore rapidement en mois d'itérations on reste bloqué dans un moins bon minimum local lorsqu'une optimisation fine est requise. La méthode hybride quant à elle est en théorie une stricte amélioration de la méthode basique car dans le pire des cas elle ne change pas la solution de base.

4 Conclusion

La complexité du problème implique des voisinages de grandes tailles et demande de l'efficacité cependant les contraintes dures demandent des mécanismes complexes pour les vérifier qui imposent une dégradation en qualité de la solution ou en temps de calcul. La solution retenue la GVNS avec les voisinages d'interchange puis 2-swap, le shake et le décodage hybride. On peut l'adapter pour les plus petites instances en privilégiant les mouvements plus petit et le décodage de base.

5 Références

[1] Cheng, R. , Gen, M., and Tsujimura, Y. :A Tutorial Survey of Job Shop Scheduling Problems Using genetic Algorithms-I. Representation. Journal of Computers and Industrial Engineering 30(4) (1996) 983-997

[2] B. Norman and J. Bean. Random keys genetic algorithm for job-shop scheduling: unabridged version. Technical report, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor (1995).
[3] Chapter 3 : Variable Neighborhood Search, Pierre Hansen, Nenad Mladenovic, Jack Brimberg and Jos´e A. Moreno P´erez
[4] Variable Neighbourhood Search for Job Shop Scheduling Problems, Mehmet Sevkli, M. Emin Aydin