Outils pour la résolution exacte en optimisation (2023-2024)

Le but de ces TP est de résoudre un problème d'optimisation discrète de façon approchée et exacte. Il faudra ainsi programmer une heuristique, puis une méthode par séparation et évaluation. Vous serez pour cela guidé par des exercices progressifs.

1 Langage de programmation

Le langage utilisé est Python.

2 Problème étudié : le flowshop de permutation

Dans ce problème, on dispose de m machines $M_1, \ldots, M_m$ sur lesquelles on doit exécuter un ensemble de n tâches, appelées *jobs*. Chaque job est constitué de m opérations qui doivent être exécutées sur les machines $M_1, \ldots, M_m$, dans cet ordre. La durée d'une opération sur la machine M_i pour un job j est notée p_{ij}. Une machine ne peut exécuter plus d'une opération à la fois et l'exécution d'une opération ne peut être interrompue.

On impose enfin d'avoir le même ordre de passage des jobs sur toutes les machines. Le nombre d'ordonnancements possibles correspond alors au nombre de permutations $n!$.

Par exemple, pour un problème à 4 jobs, il suffit de déterminer un ordre parmi les 4 ! pour déterminer un ordonnancement. Si on choisit l'ordre (4,1,3,2) (par ex.) sur la machine 1 on aura l'opération 1 du job 4, suivie par l'opération 1 du job 1, suivie par l'opération 1 du job 3, puis par l'opération 1 du job 2 ; sur la machine 2 on aura l'opération 2 du job 4, suivie par l'opération 2 du job 1, suivie par l'opération 2 du job 3, puis par l'opération 2 du job 2 ; et ainsi de suite.

La durée d'un ordonancement, notée $C_{\max}$, est la date de fin de l'opération se terminant le plus tard. L'objectif du problème est de minimiser $C_{\max}$.

3 Méthodes utilisées

3.1 Résolution approchée

L'une des heuristiques les plus efficaces est appelée NEH, du nom des ses auteurs Nawaz, Enscore et Ham [2]. Elle consiste à appliquer deux étapes :

1. Créer une liste des jobs triés selon l'ordre décroissant de leurs durées totales, c'est-à-dire selon les valeurs $\sum_{i=1}^{m} p_{ij}$.
2. Créer un ordonnancement ne contenant que le premier job de la liste. Ajouter ensuite chaque job, selon l'ordre de la liste, en testant toutes les positions possibles et en retenant celle qui minimise la durée de l'ordonnancement.

Exemple : *supposons que l'on ait 5 jobs et que l'ordre obtenu à l'étape 1 soit (2,4,1,5,3). A l'étape 2 on va commencer par créer un ordonnancement avec le job 2 tout seul, puis on va regarder où placer le job 4, puis 1, puis 5, et enfin 3. Pour choisir à quelle place on doit ajouter le job 4 on va tester l'ordonnancement (4,2) puis (2,4) et l'on retiendra celui qui minimise* $C_{\max}$. *Supposons que ce soit l'ordre (4,2). Pour ajouter le job 1 on procède de la même façon : on va tester l'ordonnancement (1,4,2), puis (4,1,2), puis (4,2,1) et l'on retiendra celui qui minimise* $C_{\max}$. *Et ainsi de suite.*

La première étape peut se faire en temps $O(n \log n)$ puisqu'il s'agit d'un tri. La seconde étape consiste à tester deux positions, puis trois positions, et ainsi de suite, jusqu'à tester n positions, c'est-à-dire à évaluer au total $O(n^2)$ ordonnancements. Puisque chaque ordonnancement peut être évalué en $O(mn)$ on arrive à une complexité finale en $O(mn^3)$.

3.2 Résolution exacte

Chaque permutation permet de définir un ordonnancement et l'ensemble des permutations correspond à l'ensemble des ordonnancements possibles. On utilise donc une séparation permettant de générer toutes les permutations.

L'évaluation se fait par le calcul d'une borne inférieure assez simple [1] qui repose sur les observations suivantes :

1. Un job j ne peut démarrer sur une machine M_k avant que les opérations de j sur les machines précédentes ne soient terminées, c'est-à-dire avant la date
$$r_{kj} = \sum_{i=1}^{k-1} p_{ij}, \text{ avec } r_{1j} = 0. \tag{1}$$
2. Lorsqu'un job j est terminé sur une machine M_k il reste à exécuter les opérations de j sur les machines suivantes, ce qui représente une durée au minimum égale à
$$q_{kj} = \sum_{i=k+1}^{m} p_{ij}, \text{ avec } q_{mj} = 0. \tag{2}$$
3. Quel que soit l'ordre choisi, toutes les opérations qui doivent être exécutées sur une machine le seront, donc une machine M_k est occupée durant une durée
$$\sum_{1 \leq j \leq n} p_{kj}. \tag{3}$$

L'observation 1 implique qu'une machine M_k sera disponible au plus tôt à la date $\min_{1 \leq j \leq n} r_{kj}$ (appelée *date au plus tôt*). De même, l'observation 2 implique qu'après l'exécution du dernier job sur M_k il s'écoulera au minimum une durée $\min_{1 \leq j \leq n} q_{kj}$ (appelée *durée de latence*) avant la fin de l'ordonnancement. On peut alors définir un minorant **pour chaque machine** en posant

1. On pourra consulter l'article [1] pour une revue de bornes plus sophistiquées et plus efficaces.

$$LB_k = \min_{1 \leq j \leq n} r_{kj} + \sum_{1 \leq j \leq n} p_{kj} + \min_{1 \leq j \leq n} q_{kj}. \quad (4)$$

La valeur de LB_k représente donc la plus petite date à laquelle **peut** se terminer un job sur la machine k.

Si l'on considère toutes les machines on peut déduire qu'aucun job ne peut se terminer avant la valeur $\max_{1 \leq k \leq m} LB_k$. En posant $LB = \max_{1 \leq k \leq m} LB_k$ on a donc

$$LB \leq C_{\max}. \quad (5)$$

La valeur LB est alors un minorant pour le problème.

4 Exercices

Les énoncés se trouvent dans le fichier `flowshop.py`.

Etudiez les fonctions pour manipuler un *job* et un *ordonnancement*, avant de démarrer l'exercice n°1.

Il est essentiel de tester votre code après chaque exercice avant de passer au suivant. Deux programmes de test sont fournis pour tester les fonctions de manipulation d'un *job* et d'un *ordonnancement*. Inspirez-vous de ces exemples pour écrire vos propres tests.

Références

[1] T. Ladhari and M. Haouari. A computationial study of the permutation flow shop problem based on a tight lower bound. *Computers & Operations Research* 32 (2005) 1831-1847.

[2] M. Nawaz, E. E. Jr. Enscore and I. Ham. A heuristic algorithm for the m-machine, n-job flowshop sequencing problem. *Omega* 11 (1983) 91-95.