EILCO INFO3 — Mineure IA

Résolution de problèmes – TP

Octobre 2025

Le problème du flowshop de permutation (FSP) est un problème d'ordonnancement très étudié dans la littérature, avec beaucoup de variantes dont la plupart sont NP-complètes. On s'intéresse ici au problème à $m \in \mathbb{N}$ machines avec pour objectif la minimisation de la date de fin d'ordonnancement.

Une instance de FSP est composée de n tâches (jobs) J_1 à J_n à traiter, m machines M_1 à M_m, où chaque tâche doit être traitée dans un ordre imposé, et un ensemble de $n \times m$ tâches t_{ij}, où t_{ij} représente le temps de traitement de la tâche J_i sur la machine m_j. Deux tâches ne peuvent pas être traitées simultanément sur une machine et dans la variante du problème considérée, toutes les tâches doivent être traitées dans le même ordre sur chaque machine (on parle de flow-shop de *permutation*). Chaque tâche est programmée à une date s_{ij}.

Les instances utilisées sont produites à partir du générateur proposé par Éric Taillard en 1993. Le nombre de machines considéré est $m \in \{5, 10, 20\}$ et le nombre de tâches considérées est $n \in \{20, 50\}$. Les instances sont de la forme suivante :

```
20 % nombre de jobs
5 % nombre de machines
873654221 % seed (ignorer)
0 % job 0 (identifiant)
468 % (ignorer)
54 79 16 66 58 % duree du job 0 sur chaque machine : t01, t02, t03, t04, t05
1 % job 1
325 % (ignorer)
83 3 89 58 56 % t11, t12, t13, t14, t15
2 % job 2
923
15 11 49 31 20 % t21, t22, t23, t24, t25
3 % job 3
513
71 99 15 68 85 % t31, t32, t33, t34, t35
4 % job 4
1070
77 56 89 78 53 % t41, t42, t43, t44, t45
...
```

Ici la fonction objectif à optimiser correspond à la date de fin d'ordonnancement (appelée makespan). La fonction objectif considérée C_{max}, à minimiser est décrite comme suit :

$$C_{max}(\Pi) = max_{i \in [1,n]}\{s_{im} + t_{im}\}$$

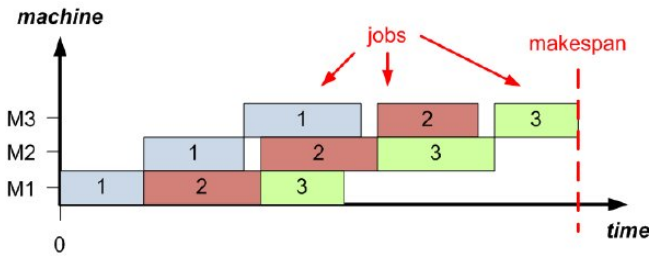

FIGURE 1 – Exemple d'ordonnancement avec 3 jobs et 3 machines.

Question 1

Créer une fonction qui charge en mémoire les données (utiles) d'une instance du problème. Attention, votre programme devra être capable de prendre en argument le nom/chemin de l'instance à charger.

Question 2

Écrire une fonction qui permet de générer aléatoirement une solution valide. Une solution contient les n tâches, une seule fois et correspond à l'ordre dans lequel elles sont exécutées sur les différentes machines.

Question 3

Écrire une fonction `cout_CMax` qui retourne le coût d'une solution (fonction objectif). Pensez à vérifier la validité de vôtre fonction avec des petites instances exemples, créées à la main.

Question 4

Écrire une fonction `echange`, qui modifie une solution en échangeant la position de deux jobs (ex : 87654321 devient 83654721 si l'on echange le jobs en position 2 en position 6). La fonction prend donc en compte deux entiers en paramètre.

Question 5

Écrire une fonction `insere`, qui modifie une solution en déplaçant un job à une nouvelle positon (ex : 16824537 devient 18245637 si l'on insère les jobs en position 2 et 6). La fonction prend donc en compte deux entiers en paramètre, l'un pour la position du job à déplacer, l'autre pour indiquer sa nouvelle position.

Question 6

Écrire une fonction `marche_aleatoire`, qui modifie aléatoirement une solution via un opérateur de voisinage. A chaque ps de la marche, on effectue un mouvement choisit aléatoirement dans le voisinage. La marche s'arrête après n d'évaluations (quand $n = 1million$, par exemple). On retourne alors la meilleure solution rencontrée (et l'évaluation de son coût).

Question 7

Écrire une fonction `climber_first`, qui s'arrête dès qu'un optimum local strict est atteint (on retourne alors la dernière solution obtenue). Faire la même chose pour `climber_best`.
Indication : les voisins doivent être générés dans un ordre aléatoire. Pour cela, on génère en début d'exècution l'ensemble des paires de positions possibles, ; ensuite on choisit aléatoirement dans cet ensemble, sans remise des voisins choisis.

Question 8

Proposer votre proppre algorithme, qui ne s'arrête pas systèmatiquement une fois un optimum local atteint, mais après un certain nombre d'évluations (par exemple 1 million).

Question 9

Écrire un programme de test, qui exécute chacun des 4 algorithmes : la marche aléatoire, les deux climbers, et votre méthode. En considérant les deux voisinages possibles, cela fait donc 8 versions à expérimenter.

Ce programme prend en paramètre le nombre k d'exécutions par méthode, et le nombre n de solutions évaluées pour chaque exécution.

À chque fois qu'une variante aura été testée k fois, le programme affichera le coût moyen atteint, ainsi que le nombre moyen d'évaluations.