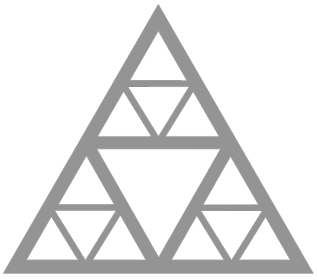

École des Ponts

ParisTech

PROJET DE RECHERCHE OPÉRATIONNELLE

Mohammed HAMZAOUI, Arthur Bresnu, Guillaume HENON-JUST

École Nationale des Ponts et Chaussées, Champs-sur-Marne, France

Groupe 11

1 Linéarisation du Problème

Après plusieurs tentatives de linéarisation la contrainte (7) du sujet, nous sommes parvenus à une approche avec des indicatrices $x_{i,m,o}$ et une valeur absolue de différence qu'on impose strictement positive pour assurer la non-égalité des termes. Ceci nous amène à la linéarisation suivante :

1.1 Première linéarisation

$$\min_{Bi,m_i,o_i} \quad \sum_{j\in J} w_j(C_j + \alpha U_j + \beta T_j) + \sum_{i,m,o} x_{i,m,o} + \sum_{i,i'} \eta_{ii'} \sum_{i,i',o} \eta^o_{ii'} + \sum_{i,i',m} \eta^m_{ii'}$$

(1)	$C_i = B_i + p_i$	$\forall i \in I$
(2)	$B_j = B_{i_1}$	$\forall j \in J$
(3)	$C_j = C_{i_{kj}}$	$\forall j \in J$
(4)	$B_j \geq r_j$	$\forall j \in J$
(5)	$B_{ih} \geq C_{i_{h-1}}$	$\forall i \in I, \forall h \in [\![2\,;\,k_j]\!]$
(6)	$T_j \geq 0$	$\forall j \in J$
(7)	$T_j \geq C_j - d_j$	$\forall j \in J$
(8)	$U_j \geq \frac{T_j}{\Theta}$	$\forall j \in J$
(9)	$x_{i,m,o} \leq \delta_{imo}$	$\forall i \in I, \forall m \in M, \forall o \in O$
(10)	$\eta^m_{ii'} \geq \sum_o x_{i,m,o} + x_{i',m,o} - 1$	$\forall i \neq i' \in I$
(11)	$\eta^o_{ii'} \geq \sum_m x_{i,m,o} + x_{i',m,o} - 1$	$\forall i \neq i' \in I$
(12)	$\eta_{ii'} \geq \eta^m_{ii'}$	$\forall i \neq i' \in I$
(13)	$\eta_{ii'} \geq \eta^o_{ii'}$	$\forall i \neq i' \in I$
(14)	$z_{ii'} \in \{0,1\}$	$\forall i \neq i' \in I$
(15)	$B_{i'} - (B_i + t) > \eta_{ii'} - 1 - \Pi z_{ii'}$	$\forall i \neq i' \in I, \forall t \in [\![0, p_i - 1]\!]$
(16)	$-(B_{i'} - (B_i + t)) > \eta_{ii'} - 1 - \Pi(1 - z_{ii'})$	$\forall i \neq i' \in I, \forall t \in [\![0, p_i - 1]\!]$

Avec les constantes :

- δ est un tableau 3D où δ_{imo} vaut 1 si la tâche i peut être faite par la machine m et l'opérateur o
- $\Theta > \sup_{j\in J} T_j$
- $\Pi > 100$

Les contraintes (10) à (16) permettent de linéariser la contrainte (7) du sujet. On a :

- $x_{i,m,o} = 1$ si la tâche i est faite par la machine m et l'opérateur o, $= 0$ sinon.
- $\eta^m_{ii'} = 1$ si les tâches i et i' sont faites par la machine m, $= 0$ sinon.
- $\eta^o_{ii'} = 1$ si les tâches i et i' sont faites par la machine o, $= 0$ sinon.
- $\eta_{ii'} = 1$ si les tâches i et i' sont faites par une même machine ou un même opérateur, $= 0$ sinon.
- $z_{ii'}$: variable binaire qui permet de linéariser la contrainte : $|B_{i'} - (B_i + t)| > \eta_{ii'} - 1$, $\forall t \in [\![0, p_i - 1]\!]$

1.2 Symétries

On remarque les symétries suivantes :

- $\eta^m_{i,i'} = \eta^m_{i',i}$
- $\eta^o_{i,i'} = \eta^o_{i',i}$
- $\eta_{i,i'} = \eta_{i',i}$

Il suffit de les définir $\forall i, i' \in I$, $i' > i$ et non pas $\forall i \neq i'$ pour éviter ces symétries.

2 Heuristque : Algorithme Glouton

Notre solveur linéaire MILP appliqué sur la linéarisation précédente n'a convergé que sur les instances tiny et small. Sur les instances de taille plus importante, le problème était trop complexe. Nous avons donc décidé d'implémenter une heuristique afin de déterminer dans un premier temps pour chaque instance une solution. Nous avons alors opté pour un algorithme glouton.

2.1 Algorithme glouton sur les poids des jobs

Dans un premier temps, nous avons décidé d'optimiser la solution du problème selon les poids w_j des solutions. Nous avons alors appliqué l'algorithme suivant :

1. On trie les jobs j par w_j décroissant
2. Pour chaque job j choisi dans cet ordre, on regarde la séquence de tâches S_j correspondante
3. Pour chaque tâche i_k prise dans l'ordre, on regarde l'ensemble des couples (m,o) qui peuvent faire cette tâche
4. On regarde pour chacun de ces couples la première date à partir de laquelle on peut faire cette tâche avec ce couple. Cette date doit donc ètre plus grande que r_j si $k=1$ et plus grande que $C_{i_{k-1}}$ si $k>1$.
5. On sélectionne le couple (m,o) qui minimise la date à partir de laquelle la tâche i peut être faite.
6. On enregistre le fait que la machine m et l'opérateur o sont utilisés sur les pas de temps déterminés précédemment.

Nous avons alors obtenu une solution correcte, mais nous pensions pouvoir faire mieux. Etant donné que le résultat de notre algorithme glouton est entièrement déterminé par l'ordre dans lequel il traite les jobs j, nous avons testé cet algorithme sur différentes permutations des j. Pour l'instance tiny, il a été rapide de tester toutes les combinaisons (120) mais pour les instances plus grandes, le nombre de permutations est bien trop important (200 factorielle pour l'instance large). Nous avons donc exploré au hasard d'autres permutations des j ce qui a amélioré nos résultats mais a rapidement atteint un plafond de verre.

2.2 Algorithme glouton sur les caractéristique des tasks

Nous avons alors adapté notre algorithme glouton pour qu'il s'applique non plus sur une séquence de jobs j mais sur une séquence de tâches i. Nous avons tenté plusieurs méthodes pour ordonner ces tâches i. On définit la date hypothétique d'une tache i dans le job j comme étant $date^h_{i_k} = r_j + \sum_{l=0}^{k-1} p_{i_l}$ et date limite par $date^l_{i_k} = d_j - \sum_{l=k}^{k_j} p_{i_l}$.

On les a trié soit par date hypothétique croissante soit par date limite croissante. En cas d'égalité pour deux tâches différentes, on a regardé la quelle des deux avait soit le nombre de machines possibles le plus faible $|\mathscr{M}_i|$, soit le nombre d'opérateurs possibles le plus faible $|\mathscr{O}_{i,m\in\mathscr{M}_i|}$ soit le poids de son job w_j ou enfin une combinaison linéaire des 3.

Avec cette nouvelle méthode, on n'optimise plus sur les poids w_j des jobs mais sur les dates hypothétiques ou dates limites, avec la possibilité de traîter deux tâches de jobs différents à la suite, là où avec l'algorithme glouton précédent on traitait toutes les tâches d'un même job les unes après les autres.

Algorithm 1 GloutonSortingDate

```
Input: caracteristique de l'instance
Output: Solution
sol ← [[0,0,0]]∗nb_task
Task,date ← SortTaskByDate(Task,date)
Mtm ← [[]]*nb_machine
Oto ← [[]]*nb_operatotr
for all task ∈ Task do
    Couples_m_o ← OperatorMachineForTask(task)
    start,choose_o,choose_m ← inf(start_for_task(Couples_m_o, task, date[k], Mtm, Oto, p))
end for
for all task_same_job ∈ S[task] do
    date ← Update(task_same_job,task,start)
end for
sol[task] ← start,choose_o,choose_m
Insert([start,start+p[task]],Mtm[choose_m],Oto[choose_o])
return sol
```

- SortTaskByDate(Task,date) : Fonction qui trie les tâches selon une des méthodes présentées ci-dessus.
- Mtm : Liste de taille M contenant pour chaque machine les intervalles de temps sur lesquels elle est déjà utilisée
- Oto : Liste de taille O contenant pour chaque opérateur les intervalles de temps sur lesquels il est déjà utilisé
- OperatorMachineForTask() : Fonction qui renvoie les couples (m, o) de machine, opérateur qui peuvent faire la tâche i
- inf(start_for_task()) : Fonction qui renvoie le couple (m, o) pour lequel le tâche i peut être faite le plus tôt possible. Variante random: si il y a plusieurs couple possible pour la même date alors on en prend un au hasard.
- Update() : Fonction qui met à jour les dates hypothétiques et limites de toutes les tâches ayant le même job que celui actuellement traité en fonction de la date qui vient de lui être attribuée (B_i).
- Insert() : Fonction qui met à jour les intervalles sur lesquels les machines/opérateurs travaillent en rajoutant le nouvel intervalle de temps.

2.3 Résultats

Nous avons calculé une borne inférieure en considérant la situation (qui peut tout à fait ne pas être solution) où toutes les tâches i_k ont été faites à la date hypothetique $date^h_{i_k} = r_j + \sum_{l=0}^{k-1} p_{i_l}$.
On observe alors qu'avec notre première approche, on atteint un gap de 106015. A la fin de nos démarches pour améliorer notre algorithme glouton, nous sommes parvenus à diviser par deux ce gap pour atteindre un gap de 52108.

	Tiny	Small	Medium	Large	Total	Gap
Borne inférieure	441	4206	11142	21131	36920	
Glouton 1 selon poids des j	765	6912	38198	97060	142935	106015
Glouton aléatoire optimal	573	5935	30540	84630	121678	84758
Glouton 2 selon $date^h_{i_k}$	465	5360	22958	65041	93824	56904
Glouton 2 selon $date^l_{i_k}$	561	6131	24592	61081	92365	55445
Glouton 2 selon $date^l_{i_k}$ variante random choix de (m, o)	465	5360	22420	60783	89028	52108
MILP	465	4810				

3 Métaheuristique: Approche par Algorithme Génétique

A partir des algorithmes gloutons décrits dans la partie précédente, nous avions la possibilité de générer un nombre important de solutions mais pas de manière efficace pour chercher un optimum. Nous avons alors opté pour une approche par la métaheuristique de l'algorithme génétique en partant d'une population initiale générée par notre glouton.

3.1 Algorithme génétique direct

Dans un premier temps, nous avons appliqué l'algorithme génétique de manière très direct, en considérant le génôme d'un individu comme étant la suite des trois vecteurs B, M et O.

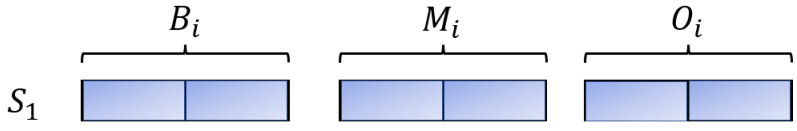

Figure 1: Croisement de deux solutions parents

Nous avons alors appliqué l'algorithme suivant en utilisant les fonctions décrites ci-dessous.

Algorithm 2 Algorithme génétique

Input: t_{max}, P_0
Output: Solution améliorée
$t \leftarrow 0$
$P_t \leftarrow P_0$
while $t \leq t_{max}$ or $c(P_{t-1}^{10\%}) \leq c(P_t^{10\%})$ **do**
  $P_t^1 \leftarrow SelectCroisement(P_t)$ {Sélection des 10 meilleures pourcents de la population }
  $P_t^c \leftarrow GenereCroisement(P_t^1)$ {Génération des enfants par croisement }
  $P_t^2 \leftarrow SelectMutation(P_t)$ {Sélection de 6 pourcents de la population }
  $P_t^m \leftarrow GenereMutation(P_t^2)$ {Génération des enfants par mutation }
  $P_{t+1} \leftarrow SelectPopulation(P_t \cup P_t^c \cup P_t^m)$ {Sélection des $|P_0|$ meilleures solutions }
  $t \leftarrow t+1$
end while

- P_0 : Population de solutions générées à l'aide de l'algorithme glouton sur des permutations de l'ordre des jobs j.
- $c(P_t^{10\%})$: Coût moyen des dix meilleurs pourcents de la population de P_t
- $GenereCroisement(P)$:

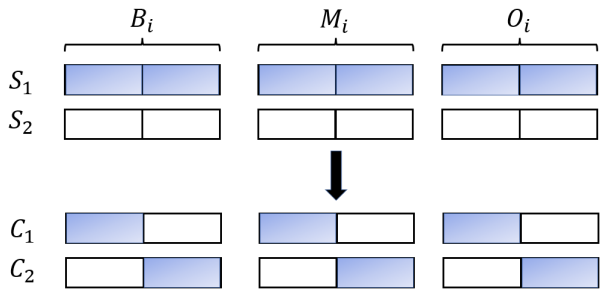

Figure 2: Croisement de deux solutions parents

- $GenereMutation(P)$: On divise la population sélectionnée en trois fois 2% : Mutation sur les machines, mutation sur les opérateurs, mutation sur les B_i.

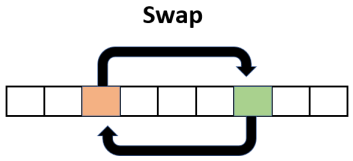

Figure 3: Mutation swap d'un vecteur B_i, M_i ou O_i

Ainsi, à chaque nouvelle génération d'enfant par mutation ou par croisement, il est nécessaire de s'assurer que cet enfant est toujours solution du problème. S'il ne l'est pas pour une mutation, on annule celle-ci et on en refait une jusqu'à trouver une solution admissible. Et c'est là que nous avons fait face à un problème majeur : le programme était très inefficace car prenait beacoup trop de temps pour générer des mutations qui soient solution du problème. De plus, les enfants par croisement étaient aussi rarement solution ce qui imposait de faire de nombreux couplages pour avoir une population de taille fixe. Le programme était donc beacoup trop lent et nous n'avons jamais pu générer une nouvelle génération de taille suffisante sur la data "large".

4 Conclusion

Nous n'avons donc pas réussi à implémenter une métaheuristique qui améliorerait les résultats de notre heuristique, mais par améliorations successives de notre algorithme glouton, nous sommes tout de même parvenus à des résultats divisant notre gap par rapport à la borne inférieure par deux.