INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

LAAS

Département de Génie Électrique & Informatique

Compte rendu

Métaheuritique - Problème du Jobshop

14 mai 2020

Etudiant :

ETSE Paul Fiagnon etse@etud.insa-toulouse.fr

Professeur :

HUGUET Marie-José

Résumé :

Ce présent document est un compte-rendu des TP de Méta-heuristique. Au cours de ces séances de TP, nous avons travaillé sur la résolution du problème de Jobshop. Pour ce faire, nous sommes partis d'un code initial contenant la modélisation du problème de Jobshop dans la représentation nommée JobNumbers et quelques méthodes de résolution. Nous y avons ajouté une représentation du nom de ResourceOrder qui range les éléments par ordre de passage sur chaque machine. À partir de la Représentation ResourceOrder, nous avons implémenté des méthodes de résolution basée sur les algorithmes gloutonnes SRP, LPT, SRPT et LRPT. Nous avons ensuite amélioré ces algorithmes en utilisant une variante qui tient compte des dates de début au plus tôt. Ce qui nous a permis d'avoir des écarts de 13%. La méthode décente et la méthode Tabou ont encore amélioré ces solutions ont donné des écarts de 10.3%. Vous pourrez trouver l'entièreté du code dans le dépôt `https://github.com/paulEtse/jobshop`[1].

Table des matières

Table des figures

Liste des tableaux

1 Prise en main du code existant

Nous allons présenter ici, le code existant via quelques diagrammes uml.

1.1 Les cas d'utilisations

L'application permet à l'utilisateur d'exécuter le problème de jobshop sur une instance donnée en choisissant une méthode de résolution bien précise comme le montre le diagramme des cas d'utilisations 1. Le code initial contenait déjà deux méthodes, la méthode basic et la méthode random. Ces deux algorithmes utilisent la représentation JobNumber.

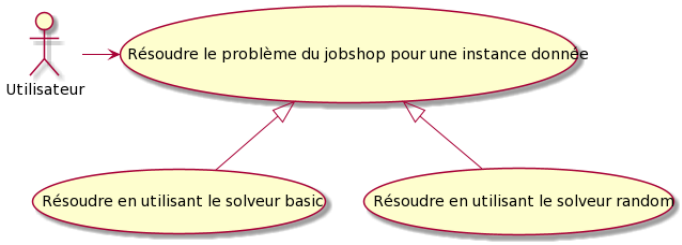

FIGURE 1 – Diagramme des cas d'utilisation

1.2 Diagramme des classes

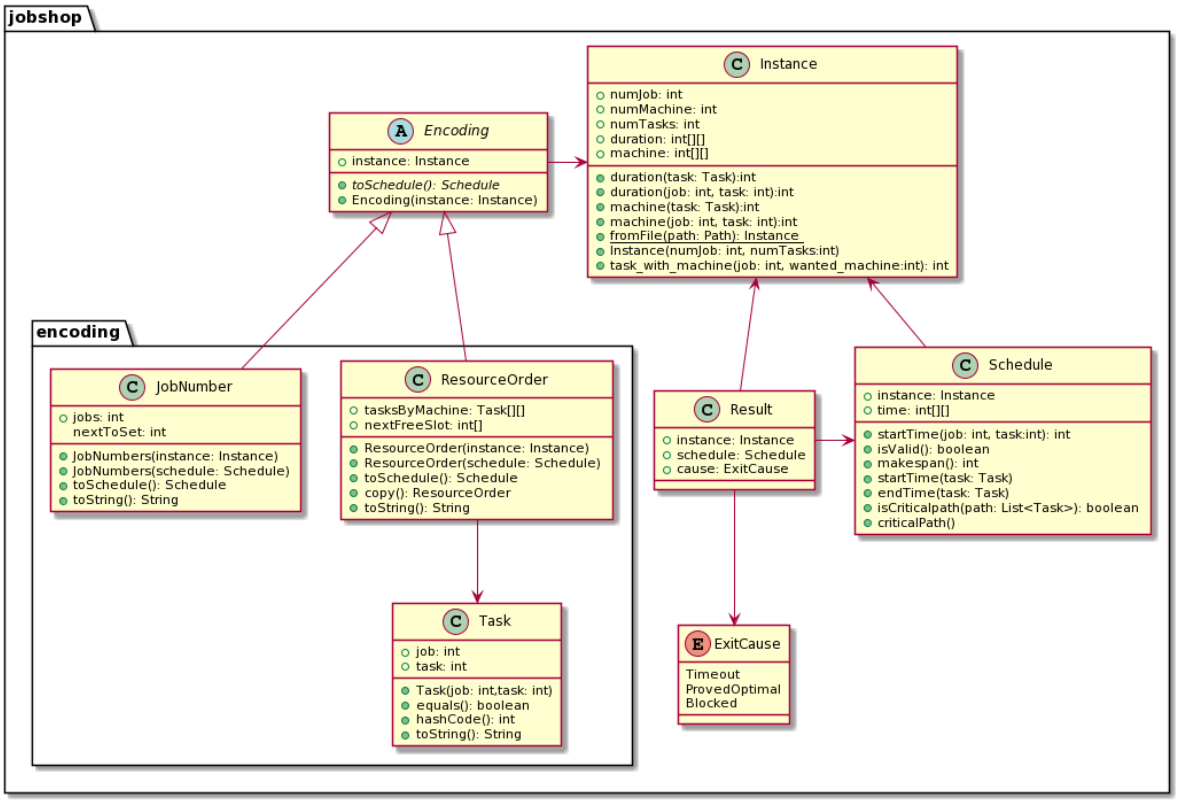

FIGURE 2 – Diagramme des classes

La classe abstrait *Encoding* permet de représenter une solution. Nous avons les deux représentations *ResourseOrder* et *JobNumber*. La classe *Instance* permet de code une instance du problème. La classe *Schedule* permet de représenter l'organisation des tâches sur les machines. Elle contient les dates de début de toutes les tâches. Elle peut aussi fournir la durée d'exécution. La classe *Result* représente une solution du problème.

L'organisation présentée dans le diagramme des classes précédent constitue une "librairie de fonctions" qui seront utilisées par les classes du la figure 3 pour la résolution du problème. L'interface *Solver* permet de représenter une méthode de résolution elle contient une fonction *solve* qui retourne une solution à partir d'une instance. Tous les solveurs implémentent cette interface. La classe Main applique les solveurs sur l'instance du problème qui lui est fourni en paramètre.

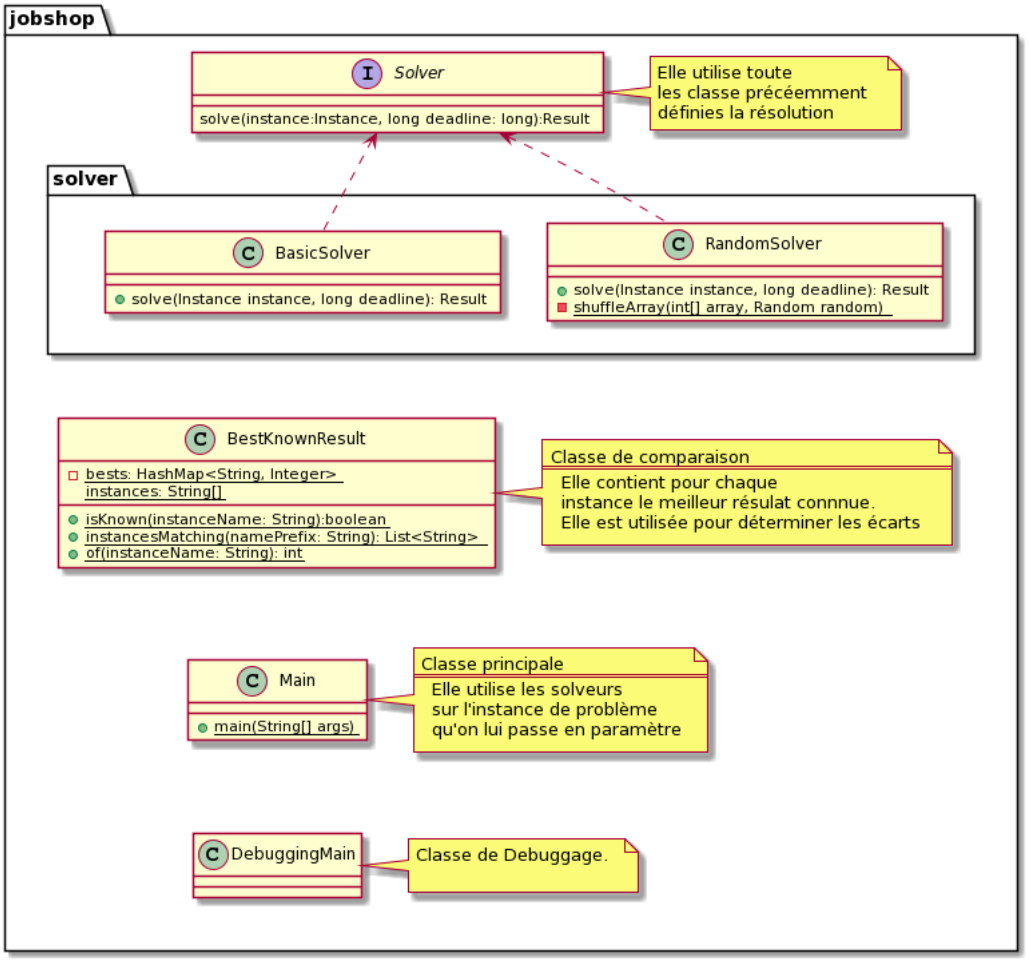

FIGURE 3 – Utilisation des classes pour la résolution du problème

2 Représentation des solutions et espace de recherche

Dans cette partie, nous allons présenter les représentations possibles d'une solution du problème de Jobshop . La taille est de l'espace de recherche dépend de la représentation considérée.

Une représentation une façon de présenter l'ordonnancement des tâches pour une solution donnée. considérons le problème du Jobshop de l'instance ft06[2] : Job Shop composé de 6 jobs et de 6 ressources. Pour chaque opération de chaque job, le tableau ci-dessous donne la ressource nécessaire et la durée de réalisation sur cette ressource. Pour résoudre le problème

J0	r2,1	r0,3	r1,6	r3,7	r5,3	r4,6
J1	r1,8	r2,5	r4,10	r5,10	r0,10	r3,4
J2	r2,5	r3,4	r5,8	r0,9	r1,1	r4,7
J3	r1,5	r0,5	r2,5	r3,3	r4,8	r5,9
J4	r2,9	r1,3	r4,5	r5,4	r0,3	r3,1
J5	r1,3	r3,3	r5,9	r0,10	r4,4	r2,1

TABLE 1 – Job shop, instance ft06

de Jobshop, il a été adoptée trois différentes représentation : JobNumber , ResourceOrder et Schedule. Nous allons présenter ici les différentes représentation.

2.1 Représentation JobNumber et espace de recherche associé

Cette représentation se base sur un ordre entre les opérations des jobs. Il serait possible de fournir un tableau contenant un ordre total de l'ensemble des opérations (une permutation sur l'ensemble des opérations). Cependant le nombre total de permutations serait plus important que le nombre de solutions intéressantes (pour l'objectif étudié). La variante considérée ici consiste à donner les ordres de passage des différents jobs (le même numéro de job est répété autant de fois qu'il y a d'opérations dans ce job).

Par exemple, considérons la solution qui consisterait à exécuter toutes les tâches de J0, J1, etc. jusqu'à J5. Cette solution est représentée par le tableau 2.

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	...

4	4	4	4	4	5	5	5	5	5	5

TABLE 2 – Exemple de solution avec JobNumber pour ft06

Soit n, le nombre de job, et n le nombre de tâche par Job. La taille de l'espace pour telle représentation qui correspond au nombre de permutations du numéro de jobs avec répétition [3] serait alors de :

$$\frac{(n*m)!}{m!^n}$$

Pour l'instance considérée dans notre exemple (ft06), le résultat des calculs donne : $2.64e^{24}$ solutions possibles.

2.2 Représentation JobNumber et espace de recherche associé

Une représentation plus synthétique consiste à donner pour chaque machine l'ordre dans lequel réaliser les différentes opérations [4]. Pour faire simple, considérons toujours la solution qui consisterait à exécuter toutes les tâches de J0, J1, etc. jusqu'à J5. Cette solution est représentée par le tableau 3 Les couples d'éléments dans les colonnes du tableau constituent

Machine 0	(1,2)	(2,5)	(3,4)	(4,2)	(5,5)	(6,4)
Machine 1	(1,3)	(2,1)	(3,5)	(4,1)	(5,2)	(6,1)
Machine 2	(1,1)	(2,2)	(3,1)	(4,3)	(5,1)	(6,6)
Machine 3	(1,4)	(2,6)	(3,2)	(4,4)	(5,6)	(6,2)
Machine 4	(1,6)	(2,3)	(3,6)	(4,5)	(5,3)	(6,5)
Machine 5	(1,4)	(2,4)	(3,3)	(4,5)	(5,4)	(6,3)

TABLE 3 – Exemple de solution avec ResourceOrder pour ft06

le numéro du job et le numéro de la tache dans le job en question.

Avec cette représentation, la taille de l'espace de recherche correspond à toutes les permutations de tâches sur les ressources. Elle est égale à

$$(n!)^m$$

où n'est le nombre de jobs et m le nombre de machines (en supposant que les tâches d'un job passent par toutes les machines). Parmi ces permutations, certaines correspondent à des solutions non réalisables[3].

Pour l'instance considérée dans notre exemple (ft06), le résultat des calculs donne : $1.39e^{17}$ solution possible.

2.3 Représentation par la date de début de chaque tâche et espace de recherche associé

avec la représentation par date de début de chaque tâche (Schedule), cette taille dépend des valeurs possibles des dates de début et du nombre de tâches. Elle est égale à

$$(Dmax)^{n*m}$$

où Dmax représente la durée maximale d'un ordonnancement (dans le pire cas, c'est la somme des durées de toutes les tâches). Parmi ces solutions, certaines ne

sont pas réalisables [3].

Pour notre instance ft06, avec une telle représentation, nous avons un espace de recherche de taille $3.99e^{82}$

2.4 Comparaison

D'après le tableau comparatif des temps d'exécution,

— D'une part on constate que les tailles de l'espace de recherche et les temps d'exécution avec la représentation ResourceOrder sont plus petits que ceux de JobNumber, et Schedule. Donc la meilleure représentation est ResourceOrder.

— Les temps d'exécution aussi sont globalement très importants. Donc les méthodes exhaustives ne sont pas adaptées à la résolution du problème de jobshop.

	Formule	taille de l'espace de recherche	Temps d'exécution (année)
JobNumber	$\frac{(n*m)!}{m!^n}$	$2.64e^{24}$	$8.46e^{7}$
ResourceOrder	$(n!)^m$	$1.39e^{17}$	4.4
Schedule	$(Dmax)^{n*m}$	$3.99e^{82}$	$1.26e^{66}$

TABLE 4 – Tableau comparatif des espace de recherche

3 Méthodes gloutonne

Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local. Dans certains cas cette approche permet d'arriver à un optimum global. Nous allons présenter ici, les méthodes gloutonnes[5] développées pour la représentation ResourceOrder.

3.1 Algorithme général

Le principe du "faire étape par étape" précédemment décrit sera utilisé le problème du Jobshop pour placer les tâches sur les machines dans la représentation ResourceOrder. On détermine la liste tâches réalisables et à chaque itération on choisit une tâche de cette liste qu'on place sur la ressource qu'elle demande à la première place libre dans la représentation ResourceOrder. Ensuite, on met à jour la liste des tâches réalisables en y ajoutant la prochaine tâche du job précédemment exécuté s'il n'est pas encore terminé. On dit qu'une tâche est réalisable si tous ses prédécesseurs ont été traités (précédences liées aux jobs). Il faut noter que chaque tâche doit apparaître exactement une fois dans l'ensemble des tâches réalisables et que toutes les tâches doivent être traitées. Pour choisir la prochaine tâche à réaliser dans la liste des tâches réalisables, nous avons utilisé différentes méthodes qui sont présentes dans la suite. Voici le pseudo-code général de nos algorithmes gloutons :

```
1  entree: Instance instance, Long deadline
2  sortie: Result
3  debut
4      res ← ResourceOrder(instance)
5      Max ← instance.numJobs * instance.numMachine
6      realisable ← []
7      for job allant de 1 a instance.numJob
8          realisable.ajouter( Tache(job,0))
9      fin
10     pour i allant de 1 a Max
11         t ← meilleur(realisable)
12         machine ← machine(t)
13         sol.tasksByMachine[machine][prochain[machine]++]= t
14         realisable.enlever(t)
15         si t.task < instance.numTask
16             realisable.add(Tache(job,t.tache+1))
17         fin
18     fin
19     retourner Result(instance,res.toSchedule(),Result.ExitCause.Timeout)
20 fin
```

Listing 1 – Pseudo code Methode glutonne

3.2 les algorithmes LPT et SPT

Les algorithmes LPT et SPT classent les tâches en fonction de leur durée d'exécution. SPT (Shortest Processing Time) donne priorité à la tâche la plus courte tandis que LPT (Longest Processing Time) donne priorité à la tâche la plus longue. Pour ces deux algorithmes, nous avons utilisé la méthode *duration* (de la classe *Instance*) pour obtenir les durées d'exécution des tâches et effectuer les comparaisons. Les différents résultats sont présentés dans le tableau 5.

3.3 les algorithmes LRPT et SRPT

Les algorithmes LRPT et SRPT classent les tâches en fonction de du temps d'exécution restant pour le job auquel appartient la tâche en question. SRPT (Shortest Remaining Processing Time) donne la priorité à la tâche appartenant au job ayant la plus petite durée restante tandis que LRPT (Longest Remaining Processing Time) donne la priorité à la tâche appartenant au job ayant la plus grande durée restante. Pour ce faire, nous avons utilisé un tableau, qui pour chaque job, contient le temps restant pour la terminer.

3.4 heuristique gloutonne randomisée

Pour l'heuristique gloutonne randomisée, le choix de la prochaine à réaliser se fait de façon aléatoire. On remarque dans le tableau 5 que l'algorithme glouton aléatoire donne de très bons résultats. Mais son temps d'exécution est maximal. C'est la seule question bonus que nous avons traité.

3.5 Amélioration des heuristiques gloutonnes - variante EST

Pour l'approche EST (Earliest Start Time), la priorité est donnée à la tâche pouvant commencer au plus tôt parmi l'ensemble des tâches réalisables. Pour ce faire, nous avons utilisé deux tableaux, l'un contenant les date de fin de tâches sur les job, et l'autre les date de fin sur les machines. La date de début d'une tâche n'est que le maximum entre ces deux dates. Lorque deux tâches ont la même date de début, la méthodes EST donne la priorité à l'une d'entre elle en utlisant les heuristiques LRPT, SPT, LPT et SRPT . Les différents résultats sont présentés dans le tableau 6.

3.6 Résultats des algorithmes gloutonnes

Pour faciliter la lecture des résultats, nous avons décidé de les présenter dans deux tableaux. Le premier contient les résultats des heuristiques gloutonnes classiques et le second le résultat de ces même heuristiques gloutonnes avec la considération du temps de début au plus tôt.

On constate que les algorithmes SPT et LPT ont presque les mêmes résultats en matière de temps d'exécution et l'écart. LPT est cas même mieux que SPT.

L'algorithme SRPT fournit de très mauvais résultats. C'est l'algorithme le moins intéressant parmi tous.

L'algorithme LRPT fournit de bonne performance.

On peut alors classer nos algorithmes comme ceci :

$$Random >> LRPT >> SPT >> LPT >> SRPT$$

	SPT			LPT			SRPT			LRPT			Random		
	R	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*
ft06	1	108	96.4	1	133	141.8	1	154	180	6	74	34.5	999	57	3.6
ft10	1	2569	176.2	0	2860	207.5	1	2657	185.7	1	1289	38.6	999	1285	38.2
ft20	1	2765	137.3	1	2603	123.4	1	2986	156.3	0	1955	67.8	999	1548	32.9
...															
la10	1	2149	124.3	1	2255	135.4	1	3193	233.3	6	1224	27.8	999	1026	7.1
la20	1	3029	235.8	1	3242	259.4	0	3881	330.3	1	1404	55.7	999	1188	31.7
la40	0	6098	399.0	1	7428	507.9	1	9061	641.5	1	1953	59.8	999	1944	59.1
...															
la01-la40 (moyenne)	0.3	4381.3	278	0.1	4568.4	292.6	0.2	5984	408.6	0.2	1568	41.6	999	1498	32.3
...															
Moyenne totale (ft-la)	0.5	4202.2	268.1	0.3	4379.8	283	0.3	5701.3	392.1	0.3	1535	42	999	1471.2	32.8

TABLE 5 – Tableau comparatifs des algorithmes gloutonnes

Les résultats obtenus avec les algorithmes gloutons lorsqu'on utilise la variante EST sont présentés dans le tableau 6. Ces résultats viennent confirmer notre classement précédent. Donc On peut classé tous nos algorithmes gloutons selon l'ordre :

$$EST\_LRPT >> EST\_LPT >> EST\_SPT >> EST\_SRPT >> LRPT >> LPT >> SPT >> SRPT$$

	SPT			LPT			SRPT			LRPT			Random		
	R	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*	*R*	*M*	*E*
ft06	1	108	96.4	1	133	141.8	1	154	180	6	74	34.5	999	57	3.6
ft10	1	2569	176.2	0	2860	207.5	1	2657	185.7	1	1289	38.6	999	1285	38.2
ft20	1	2765	137.3	1	2603	123.4	1	2986	156.3	0	1955	67.8	999	1548	32.9
...															
la10	1	2149	124.3	1	2255	135.4	1	3193	233.3	6	1224	27.8	999	1026	7.1
la20	1	3029	235.8	1	3242	259.4	0	3881	330.3	1	1404	55.7	999	1188	31.7
la40	0	6098	399.0	1	7428	507.9	1	9061	641.5	1	1953	59.8	999	1944	59.1
...															
la01-la40 (moyenne)	0.3	4381.3	278	0.1	4568.4	292.6	0.2	5984	408.6	0.2	1568	41.6	999	1498	32.3
...															
Moyenne totale (ft-la)	0.5	4202.2	268.1	0.3	4379.8	283	0.3	5701.3	392.1	0.3	1535	42	999	1471.2	32.8

TABLE 6 – Tableau comparatifs des algorithmes gloutonnes variante EST

4 Méthode descendante

4.1 Principe général

Le fonctionnement d'une méthode de descente s'appuie sur l'exploration successive d'un voisinage de solutions. On part initialement d'une solution avec l'une des différentes méthodes précédentes. On chercher ensuite les voisins de la solution précédente. Le meilleur voisin sera comparé à la solution actuelle. Si elle est améliorante elle devient la solution à partir de laquelle l'exploration se fera. On arrête l'exploration s'il n'y a pas d'amélioration ou lorsqu'un temps limite de calcul est atteint. Pour illustrer nos propos dans cette section, nous allons repartir avec l'instance ft06 présenté dans le tableau 2.

4.2 Le voisinage

Le voisinage qui va être implémenté s'appuie sur la notion de chemin critique et sur la notion de blocs dans le chemin critique. Le chemin critique pour une solution donnée, c'est une suite de tâches qui détermine la durée totale d'exécution du système. Pour une solution donnée, on peut estimer la durée totale de l'ordonnancement des tâches. Et c'est déterminé par une séquence de tâche. C'est cette suite de tâches qui forme le chemin critique. Pour plus de détail, veuillez consulter la section 5 du document [3]. Le chemin nous a été fourni dans le code de base via la fonction *criticalPath*
Un bloc est composé de tâches contiguës dans le chemin critique utilisant la même ressource.

Exemple : pour notre instance ft06 considéré, nous avons utilisé comme solution de départ, la solution obtenue avec le Solver *est_ lrpt* (voir tableaux 7).

Le chemin critique de cette solution est :

M0	(0,1)	(3,1)	(2,3)	(5,3)	(1,4)	(4,4)
M1	(1,0)	(3,0)	(5,0)	(0,2)	(4,1)	(2,4)
M2	**(2, 0)**	**(0, 0)**	**(4, 0)**	**(1, 1)**	(3, 2)	(5, 5)
M3	(2, 1)	(5, 1)	(0, 3)	(3, 3)	(1, 5)	(4, 5)
M4	**(1, 2)**	**(4, 2)**	**(3, 4)**	**(2, 5)**	**(0, 5)**	**(5, 4)**
M5	(2, 2)	(5, 2)	(0, 4)	(1,3)	(4, 3)	(3, 5)

TABLE 7 – Solution initiale sur ft06 obtenue avec est_lrpt

Ce chemin comprend deux blocs : le bloc 0-3 dont les tâche utilisent la machine 2 et le bloc

(2, 0)	(0, 0)	(4, 0)	(1, 1)	(1, 2)	(4, 2)	(3, 4)	(2, 5)	(0, 5)	(5, 4)	(5, 5)

TABLE 8 – Chemin critique

0-5 dont les tâche utilisent la machine 4.

Les voisins de cette solution sont obtenus par la permutation des deux éléments aux extrémités des blocs. Ici, on 4 voisin possibles :

- voisin obtenu par la permutation de ***(2,0) - (0,0)***
- voisin obtenu par la permutation de ***(4,0) - (1,1)***
- voisin obtenu par la permutation de ***(1,2) - (4,2)***
- voisin obtenu par la permutation de ***(0,5) - (5,4)***

4.3 Pseudo-code

4.3.1 Liste des blocks d'un chemin critique

Pour déterminer les voisins d'une solution, on tout d'abord chercher le chemin critique de cette solution via la fonction *criticalpath* qui nous a été fourni. A partir de ce chemin crittique qui est une liste de tâche, nous extrayons la liste des block via l'algorithme suivant.

```
entree: ResourceOrder order
sortie: List<Block>
debut
    chemin_critique ← chemin_critique(order)
    blocks ← []
    i ← 0
    Tmax ← nb_elts_dans(chemin_critique)
    tant que i < Tmax
        t ← chemin_critique[i]
        machine ← machine(t)
        j ← i+1
        tant que j < Tmax ET machine(chemin_critique[j]) = machine
            j++
        fin
        si j > i+1
            premier = chemin_critique[i].indice
            dernier = chemin_critique[j-1].indice
            blocks.ajouter(Block(machine, premier, dernier))
        fin
        i ← j
    fin
    retourner blocks
fin
```

Listing 2 – Chemin critique d'une solution

4.3.2 Voisinage d'une solution

Le voisinage d'une solution correspond à l'ensemble des voisins obtenus par permutation dans chaque bloc. Pour cela la structure de donnée Swap nous a été fourni. Les voisins sont obtenu par permutation des éléments aux extrémités donc pour tout bloc comportant deux tâches, on a seule solution voisine et pour tout bloc ayant au moins trois tâches, on a deux solutions voisines. L'algorithme de détermination des voisins est le suivant :

```
entree: Block bloc
sortie: List<Swap>
debu
    swaps ← []
    swaps.ajouter(Swap(machine(block),premier(block),premier(block)+1))
    si dernier(block) > premier(block)+1
        swaps.ajouter(Swap(machine(block),dernier(block)-1,dernier(block));
    fin
    retourner swaps
fin
```

Listing 3 – Voisins d'une solution

4.3.3 Algorithme principale

```
entree: Instance instance, Long deadline
sortie: Result
debut
    res ← solution_de_est_lrpt()
    change ← Vrai
    tant que change ET deadline_non_atteint
        change = Faux
        meilleur=meilleur_voisin(res);
        si temp_total(meilleur) < temp_totale(res)
            res=meilleur;
            change=Vrai;
        fin
    fin
    retourner Result(instance,res.toSchedule(),Result.ExitCause.Timeout)
fin
        ____________________________________________________________
Le meilleur voisin est une fonction qui calcul l ensemble des voisin et
   retourne le meilleur celui dont le temps d execution est le plus petit
```

Listing 4 – Pseudo code Methode descendante

4.4 Comparaison des résultats

Par rapport aux résultats obtenus avec les méthodes précédentes, on peut dire que la méthode descente donne meilleur au niveau du temps d'exécution, du makespan et donc au niveau des écarts aussi. Son écart moyen est de 10.06% tandis que la méthode *est_lrpt* qui donnait les meilleurs résultats a un écart de 13.06. Le tableau 9 ci-dessous présente les résultats du lrpt.

	Méthode descendate		
	Makespan	***RunTime***	***Ecart***
ft06	55	27	0
ft10	1108	5	19.1
ft20	1501	4	28.8
...			
la10	996	22	0.8
la20	964	5	6.9
la40	1405	22	15
...			
la01-la40 (moyenne)	1216.5	6.7	10.2
...			
Moyenne global (ft la)	1193	6.5	10.6

Table 9 – Résultats de la méthode descendante

5 Méthode Tabou

5.1 Principe général

La méthode Tabou est une méta-heuristique basée sur l'exploration de voisinage et permettant de sortir des optima locaux en acceptant des solutions non-améliorantes. Elle fonctionne de la même façon que la méthode descente, mais au niveau de prochaine solution à développer, elle part toujours du meilleur voisin même si celle-ci n'est pas améliorante. De plus, afin d'éviter de boucler sur des solutions déjà visitées, elle garde en mémoire (pendant un certain temps), la liste des solutions déjà visitées afin d'éviter de les considérer de nouveau.

On part toujours d'une solution initiale obtenue avec l'une des méthodes précédentes (nous, nous avons choisi le EST_LRPT car c'est la méthode qui donne les meilleurs parmi nos méthodes gloutonnes.), cette solution sera considérée comme étant la solution optimale et la solution courante. On cherche ensuite les voisins de cette solution, tout en gérant les solutions tabou, et on développe à partir du meilleur voisin non-tabou ; ce traitement sera répété au bout d'un certain nombre d'itérations.

5.2 Pseudo-code

5.2.1 Le voisinage

Nous avons utilisé le même voisinage qu'au niveau de la méthode descendante. Donc on n'a pas eu besoin de ré-implémenter le voisinage. Ce qui change vraiment, c'est le choix de la prochaine solution à développer et la gestion des solutions tabou qui sera présentée dans les lignes à venir.

5.2.2 Solutions Tabou

Pour gérer les solutions Tabou, nous avons développé une structure de donnée permettant de savoir si une solution a été déjà visitée ou non. Pour des raisons d'efficacité de cette vérification, il a été décidée que cette structure mémorise les changements (mouvements) interdits dans l'exploration d'un voisinage plutôt que les solutions déjà visitées.

Sur l'exemple du ft06 que nous traitions précédemment (voir tableaux 7) avec la méthode descendante, nous avions trouver que pour la première itération, on pourrait avoir 4 voisins :

- voisin obtenu par la permutation de ***(2,0) - (0,0)***
- voisin obtenu par la permutation de ***(4,0) - (1,1)***
- voisin obtenu par la permutation de ***(1,2) - (4,2)***
- voisin obtenu par la permutation de ***(0,5) - (5,4)***

Pour chacun de ces voisins la méthode tabou choisi la meilleure permutation et interdit l'autre. Cette interdiction sera effective pendant une certaine durée. Pour notre exemple, 4 entrées seront ajoutées à la structure Tabou (voir tableau 10). On trouve que :

- ***(1,1) - (4,0)*** interdit *(4,0) - (1,1)*
- ***(4,2) - (1,2)*** interdit *(1,2) - (4,2)*
- ***(2,0) - (0,0)*** interdit *(0,0) - (2,0)*
- ***(5,4) - (0,5)*** interdit *(0,5) - (5,4)*

La structure permettant de gérer les solutions Tabou est composée d'une matrice dont les indices sont des tâches et chaque case contient l'itération à partir de laquelle une solution donnée n'est plus tabou. Cette structure donnée offre des opérations comme ajouter (qui

	...	*(1,1)*	*(1,2)*	...	*(2,0)*	...	*(5,4)*	...
(0,0)					6			
...								
(0,5)							6	
...								
(4,0)		6						
...								
(4,2)			6					
...								

Les cases vide du tableau contiennent des zéros.

TABLE 10 – Matrice Tabou de l'instance ft06 à la première it

permet d'ajouter à la structure une nouvelle solution Tabou), est_tabou qui permet de tester si une solution est Tabou. Son pseudo code est le suivant :

```
Donnees:
List<Tache> taches
Matrice_d_entier[][] matriceTabou
Entier dureeTabou


Operations:
Operation Initialiser
entree: Entier nbJob, Entier nbTache, Entier duree
sortie:
debut
    taches ← []
    pour j allant de 1 a nbJob
        pour t allant de 1 a nbTache
            tache ← Tache(j,t)
            taches.ajouter(tache)
            matriceTabou[taches.indice(tache)][taches.indice(tache)] ← 0
fin

Operation ajouter
entree: Tache t1, Tache t2, Entier iteration
sortie:
debut
        matriceTaboo[t1.indice][t2.indice] ← dureeTabou + iteration
fin

Operation est_tabou
entree: Tache t1, Tache t2, Entier iteration
sortie: Vrai / Faux
debut
    returner iteration < matriceTabou[taches.indice(t1)][taches.indice(t2)]
fin
```

Listing 5 – Structure de donnée Tabou

5.2.3 algorithme principal

```
entree: Instance instance, Long deadline
```

```
sortie: Result
debut
    Sinit ← solution_de_est_lrpt()
    tabou ← Structure_Tabou
    kMax ← 15
    dureeTabou ← 5 (par exemple)
    meilleur = sInit
    courant = sInit
    int k=0;
    tant que k < kMax ET deadline_non_atteint
        courant = meilleur_voisin(courant);
        si temp_total(meilleur) < temp_totale(courant)
                meilleur = courant
        fin
    fin
    retourner Result(instance,res.toSchedule(),Result.ExitCause.Timeout)
fin
```

Listing 6 – Méthode Tabou

Pour le choix du meilleur voisin, nous utilisons l'algorithme suivant :

```
entree: ResourceOrder s, Tabou sTaboo, Entier k
sortie: ResourceOrder
debut
    blocks ← block_du_chemin_critique(s)
    swaps ← []
    pour block appartenant a block
        swaps.ajouter(voisins(block))
    fin
    # Nous avons recupere ici tous les voisins
    premier ← Vrai
    pour swap appartenant a swaps
        sol_courant ← copie(s)
        t1 ← Tache_d_indice(swap.premier)
        t2 ← Tache_d_indice(swap.second)
        si(est_tabou(t1,t2,k) == Faux)
            swap.appliquer_permutation(sol_courant)

            # Gestion de la solution
            si premier
                meilleur ← sol_courant
                premier ← Faux
            sinon si temp_total(sol_courant) < temp_total(meilleur)
                    meilleur ← sol_courant;
            fin

            # Gestion du Tabou
            si temp_total(sol_courant) < temp_total(s)
                sTabou.ajouter(t2,t1,k)
            sinon
                sTabou.ajouter(t1,t2,k)
            fin
        sinon
            swap.appliquer_permutation(sol_courant)
            si temp_total(sol_courant) < temp_total(meilleur)
                # Solution Tabou ameliorant
                meilleur ← sol_courant
            fin
```

```
38            fin
39        fin
40        retourner meilleur
41    fin
```

Listing 7 – Choix du meilleur voisin

5.3 Résultat et comparaison

Les résultats de la méthode tabou sont présentés dans le tableau 11. On peut remarquer par comparaison aux résultats de la méthode descente qu'elle donne de meilleurs résultats. Mais au niveau du temps d'exécution, la méthode descente est meilleur.

Il est à noter que nous avons choisi comme nombre d'itérations 15 et comme duréeTabou 5. Nos résultats auront été meilleurs si ces valeurs sont plus grandes moyennant aussi de plus longs temps d'exécution.

	Tabou		
	Makespan	***RunTime***	***Ecart***
ft06	55	36	0
ft10	1108	23	19.1
ft20	1501	24	28.8
...			
la10	996	41	0.8
la20	964	20	6.9
la40	1393	63	14
...			
la01-la40 (moyenne)	1213.3	15.3	9.9
...			
Moyenne global (ft la)	1190.6	14.9	10.3

Table 11 – Résultats de la méthode Tabou

6 Comparaison des différentes méthodes

Dans le tableau 12, nous présentons la comparaison de toutes les méthodes par ordre croissant de performance. Ces résultats montrent que :

$$Tabou >> Descendant >>$$
$$EST\_LRPT >> EST\_LPT >> EST\_SPT >> EST\_SRPT >>$$
$$LRPT >> LPT >> SPT >> SRPT$$

la méthode Tabou donne les meilleurs résultats, mais avec temps d'exécution un peu plus long que celui de la méthode déscendante. Donc s'il faut faire le rapport résultat sur temps d'exécution, la méthode déscende serait meilleurs. *Il est à noter que pour la méthode Tabou, nous avons choisir pour nos tests, 15 comme nombre d'itérations et 5 comme duréeTabou. Plus ces nombres sont élevés, plus on a de meilleurs résultats moyennant aussi un temps de calcul importante.* Après eux, nous avons les méthodes EST qui donnent de bon résultats.

		Résultats moyen sur ft et la		
Rang	***Algotithme***	makespan	RunTime	Ecart
1	***Tabou***	1190.6	14.7	10.3
2	***Descendant***	1193.6	5.8	10.6
3	***Est_ lrpt***	1216.9	0.4	13
4	***Est_ spt***	1308.8	0.3	21.7
5	***Est_ lpt***	1432.4	0.4	32.2
6	***Est_ srpt***	1461.3	0.3	35.8
7	***lrtp***	1535.8	0.5	42
8	***lpt***	4202.2	0.3	268.1
9	***spt***	4379.8	0.3	283.2
10	srpt	5701.3	0.2	392.1

Table 12 – Tableau comparatif des résultat des différents algorithmes

Est_ lrpt est le meilleur parmi les différentes méthodes EST. Ce qui est un peu surprenant, c'est l'écart entre les performances du *Est_ lrpt* et celles du *Est_ srpt* ; *Est_ lrpt* est largement meilleur que *Est_ srpt*. Donc c'est mieux de commencer par les tâches ayant la durée d'exécution restante les plus longues. Enfin, nous avons les résultats des méthodes gloutonnes basique. LRPT est le meilleur parmi elles.

Les tableaux annexes ci-dessous présentent les résultats obtenus avec d'autres instances. Il confirme les constats précédents. On remarque qu'au niveau des instances orb, SRPT un peu meilleur que LRPT. Mais Globalement, notre ordre reste respecté.

		Résultat moyen pour toutes les instances a		
Rang	***Algotithme***	makespan	RunTime	Ecart
1	***Tabou***	784.7	37.3	11.6
2	***Descendant***	791	13.2	12.3
3	***Est_ lrpt***	802.8	2.2	13.8
4	***Est_ spt***	838.2	0.7	18.5
5	***Est_ lpt***	936.5	1	29.8
6	***Est_ srpt***	998.5	0.7	38.1
7	***lrtp***	982.3	1.7	37.6
8	***lpt***	3260.3	0.8	337.2
9	***spt***	3391.8	0.8	349.7
10	**srpt**	4643.3	0.5	515.7

Table 13 – Résulat comparatif pour les instances a

		Résultat moyen pour toutes les instances orb		
Rang	***Algotithme***	makespan	RunTime	Ecart
1	***Tabou***	1085.4	20.01	19.9
2	***Descendant***	1089.8	9.2	20.4
3	***Est_ spt***	1128.8	0.5	25.2
4	***Est_ lrpt***	1131.6	0.8	25.2
5	***Est_ lpt***	1230.1	0.4	35.1
6	***Est_ srpt***	1212.2	0.1	33.7
7	***lrtp***	1425.3	1.2	58.3
8	***lpt***	2460.5	0.7	176.7
9	***spt***	2922.1	0.5	228.3
10	**srpt**	3317.2	0.4	277.4

Table 14 – Résulat comparatif pour les instances orb

		Résultat moyen pour toutes les instances ta		
Rang	***Algotithme***	makespan	RunTime	Ecart
1	***Tabou***	2296.6	76	18.6
2	***Descendant***	2301.7	33.2	18.9
3	***Est_ lrpt***	2335.1	1.2	20.9
4	***Est_ spt***	2495	1	29.2
5	***Est_ lpt***	2814	0.9	45.3
6	***Est_ srpt***	2861.2	0.9	48.1
7	***lrtp***	3147.2	1.1	61.9
8	***lpt***	14509.5	0.9	618.3
9	***spt***	15846.9	0.9	680
10	**srpt**	21355.1	0.7	949.6

Table 15 – Résulat comparatif pour les instances ta

Conclusion

Cette série de TP nous a permis de mettre en pratique les connaissance acquise au cours des séances de cours de méta-heuristiques sur un cas concret d'application, le problème du jobshop. Nous sommes partis d'un code du problème avec la représentation JobNumber. Nous y avons ajouté une autre représentation ResourceOrder. Nous avons ensuite implémenté de nouveaux solveurs avec notre nouvelle modélisation ResourceOrder. Nous sommes parti d'un projet contenant deux solveurs, nous y avons ajouté onze : du SRPT qui se base sur des méthodes gloutonnes jusqu'au Tabou qui utilise le principe du voisinage. La comparaison de nos différentes solveurs en terme de performance nous donne ceci :

$$Tabou >> Descendant >>$$
$$EST\_LRPT >> EST\_LPT >> EST\_SPT >> EST\_SRPT >>$$
$$LRPT >> LPT >> SPT >> SRPT$$

Nous tenons à remercier nos différents profs qui nous ont toujours soutenu tout au long des séance de TP. En effet, ces TP ont eu lieu durant le confinement dû à la crise du Covid-19. D'autre part, ce travail n'aura pas été aussi simple si nous avions pas accès aux différentes ressources cité en Référence. Nous profitons pour remercier tous les auteurs de ces ressources.

Références

[1] P. ETSE, "Heuristic methods for jobshop scheduling," https://github.com/paulEtse/jobshop, 2020.

[2] A. Bit-Monnot and V. Artuori, "Heuristic methods for jobshop scheduling," https://github.com/insa-4ir-meta-heuristiques/template-jobshop, 2020.

[3] M.-J. Huguet and A. Bit-Monnot, "Méthodes approchées pour la résolution de problèmes d'ordonnancement," *Insa Toulouse*, pp. 1–11, 2020. [Online]. Available : https://homepages.laas.fr/huguet/drupal/sites/homepages.laas.fr.huguet/files/u78/TP_Meta.pdf

[4] M.-J. Huguet, "Méthodes approchées pour la résolution de problèmes d'ordonnancement," *Laas CNRS*, pp. 1–7, 2020. [Online]. Available : https://homepages.laas.fr/huguet/drupal/sites/homepages.laas.fr.huguet/files/u78/2019-2020-exercices-Cours-TD.pdf

[5] A. Caumond, "Le problème de jobshop avec contraintes : modélisation et optimisation," Ph.D. dissertation, Université Blaise Pascal, Clermont-Ferrand II, 2006.