Rapport Travaux Pratiques Métaheuristique

Gehin Clement

15 Mai

1 Introduction

Dans le cadre de l'UF "Systèmes Intelligents" nous avons développé aux cours des scéances de travaux pratiques différentes heuristiques dans le but de résoudre le problème de JobShop. Le but étant de déterminer un ordre optimal (optention d'un temps de réalisation minimal) pour les différentes tâches sur les différentes machines. Pour cela nous avons codé différents types d'heuristique : gloutonne, à voisinage et taboo. Vous pouvez trouver le code à l'adresse suivante : https://github.com/ClemG1/JobShop.

2 Représentation du problème

Dans un premier temps nous avons réfléchis à comment représenter le problème de JobShop. Deux représentations nous étaient déjà fournies : NumJobs et Schedule. NumJobs est une représentation des tâches dans leur ordre de réalisation. Schedule est une représenation de toute les tâches associées à leurs dates de débuts. Nous devions donc créer une troisième représentation, ResourceOrder, basée sur une représentation matricielle.

Dans ResourceOrder les lignes de la matrice correspondent aux différentes machines sur lesquelles les tâches sont réalisées. Les tâches sont ensuite rangées dans leur ordre d'exécution sur la machine. Ne pouvant garder l'information sur le travail associé à chaque tâches, la matrice contient donc des couples (job,task) avec job le travail correspondant et task le numéro de tâche en rapport avec le travail. Par exemple, si la première tâche à s'exécuter sur la machine 1 est la deuxième tâche du travail numéro trois, on aura dans la première case de la matrice le couple (3,2).

Enfin, nous avons développé des fonctions qui nous permettent de passer d'une représentation à l'autre. Ainsi on pourra toujours utiliser la représentation la plus intéressante en fonction de notre situation. En effet, en fonction de ce que l'on souhaite faire certaine représentation peuvent être plus simple d'utilisation que d'autre. Par exemple, si l'on souhaite connaître la date de fin d'éxécution des tâches la représentation Schedule nous y donne accès directement tandis que les autres non. Mais si l'on souhaite connaitre

l'ordre d'éxécution la représentation NumJobs est bien plus avantageuse.

3 Méthodes exhaustives et espace de recherche

Si l'on souhaite obtenir la solution optimale et non une solution approchée, les méthodes exhaustives peuvent être utilisées. Cependant, elles génèrent en général de grand temps de calcul. Nous allons calculer ces temps afin de discuter de leurs faisabilités.

En nous basant sur les formules du sujet, nous avons calculé les différentes tailles d'espaces de recherche pour les différentes représenattions pour l'instance ft06 qui possède 6 travaux et 6 tâches par travail :

- NumJob : $2,67.10^{24}$
- ResourceOrder : $1,39.10^{17}$
- Schedule : $3,99.10^{82}$

Maintenant si on suppose que l'on possède un ordinateur très performant qui génère une solution toute les nanosecondes, les temps nécessaires pour générer toutes les solutions sont les suivantes :

- NumJob : 84671 ans
- ResourceOrder : 1 jour et 15 heures
- Schedule : $1,26.10^{61}$ siècles

Tout d'abord on peut noter une grande disparitée entre les différentes représentations. Le temps de recherche des solutions varient énormément selon comment on représente le problème ce qui renforce encore plus le besoin d'avoir plusieurs représentations comme discuté dans la section 2.

Maintenant sur leur faisabilité, nous avons calculé les temps sur une instance relativement petite (seulement 36 tâches) et en supposant que l'on

possède un ordinateur puissant (1 nanoseconde par solution). Même sous ces conditions les méthodes exhaustives basées sur NumJob et Schedule sont impossible car beaucoup trop longue. Pour ResourceOrder, on pourrait se permettre de la réaliser si l'on a pas d'autre choix, 1 jour et 15 heures restant relativement raisonnable. Mais si l'on a pas d'obligation on preferrera quand même l'éviter. On l'évitera égalment pour un problème plus important.

4 Heuristique Gloutonnes

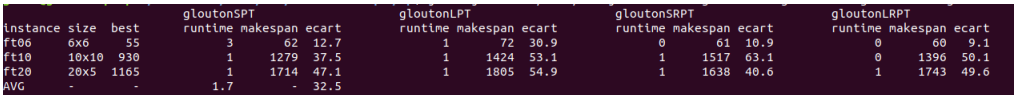
Le premier solveur que nous avons développé est un solver glouton. Le principe est de choisir la prochaine tâche à faire en fonction de différent critère. Ici nous avions quatre critères à implémenter : Shortest Processing Time (SPT), Longest Processing Time (LPT), Shortest Remaining Processing Time (SRPT) et Longest Remaining Processing Time (LRPT). Le but étant de comparer le temps total trouvé entre tout ces critères.

Pour résoudre le problème de JobShop avec une heuristique gloutonne nous avons implémenté une boucle principale : defineOrder(). Le but de cette boucle est de déterminer l'ordre des tâches en fonction de l'instance donnée en paramètre. Le fontionnement est le même pour les quatres critères, la différence réside au moment de choisir la tâche suivante ou notre choix sera orienté par le critère de selection.

Une fois l'ordre obtenu, dans la méthode solve(), on représente la solution sous forme matricielle (ResourceOrder) puis on utilise la fonction toSchedule() afin de déterminer le temps d'exécution total.

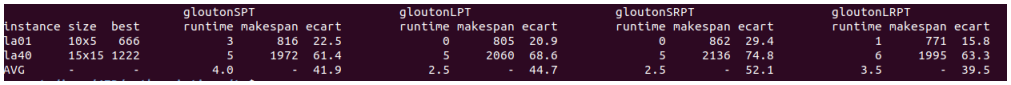
Nous avons réalisez des tests sur les quatres critères, vous trouverez sur les Figures 1 et 2 les résultats sur cinqs instances différentes. A noter que les résultats sont obtenus avec les heuristiques gloutonnes améliorées : le critère de sélection s'applique lorsque plusieurs taches on le même temps de départ sinon nous prenons celle avec le temps de départ le plus faible.

Le premier résultat que nous pouvons noter est que plus le temps nécessaire pour réaliser toute les tâches est élevé plus la différence entre le résultat obtenu et le meilleur résultat est important.



			gloutonSPT			gloutonLPT			gloutonSRPT			gloutonLRPT		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
ft06	6x6	55	3	62	12.7	1	72	30.9	0	61	10.9	0	60	9.1
ft10	10x10	930	1	1279	37.5	1	1424	53.1	1	1517	63.1	0	1396	50.1
ft20	20x5	1165	1	1714	47.1	1	1805	54.9	1	1638	40.6	1	1743	49.6
AVG	-	-	1.7	-	32.5									

Figure 1: Test avec les 4 critères sur les instances ft06, ft10 et ft20



			gloutonSPT			gloutonLPT			gloutonSRPT			gloutonLRPT		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
la01	10x5	666	3	816	22.5	0	805	20.9	0	862	29.4	1	771	15.8
la40	15x15	1222	5	1972	61.4	5	2060	68.6	5	2136	74.8	6	1995	63.3
AVG	-	-	4.0	-	41.9	2.5	-	44.7	2.5	-	52.1	3.5	-	39.5

Figure 2: Test avec les 4 critères sur les instances la01 et la40

Ensuite contrairement à ce que l'on pouvait aimaginer il n'y a pas un critère qui semble plus optimal qu'un autre. En effet, on pouvait légitimement penser qu'une stratégie pouvait être meilleure que les autres or ce n'est pas le cas. Même si le critère LRPT semble meilleur en moyenne, on a des cas comme avec l'instance ft10, où d'autres critères sont plus efficace (37.5 d'écart pour STP contre 50.1 pour LRPT).

On en conclut donc que l'efficacité de l'heuristique dépend bien évidemment du critère mais également de l'instance. Un critère sera plus efficace sur certaine instance que sur d'autre. Il semble donc nécessaire de tester plusieur critère pour une même instance afin d'obtenir un résultat le plus optimal possible.

5 Méthode de descente

Le but de cette méthode est de partir d'une solution initiale, ici trouvée grâce à l'heuristique gloutonne, et de générer ses voisins afin de trouver une solution plus optimale. Pour générer les voisins on cherche dans un premier temps des bloques dans la solution courante. Un bloque est une succession de tâches s'exécutants sur la même machine. Lorsqu'un bloque est trouvé on génère deux voisins en inversant les deux premières tâches du bloque et les deux dernières, si le bloque est de taille 2 un seul voisin est généré.

Après avoir généré tout les voisins possible à partir d'une solution on

cherche le meilleur à l'aide de la methode toSchedule() de ResourceOrder qui nous donne le temps total pour réaliser toutes les tâches. On s'arrête lorsque le meilleur voisin sélectionné est lui-même moins optimal que la solution courante.

Vous trouverez sur les Figures 3 et 4 les tests réalisés sur la méthode de descente comparée avec le résultat obtenue avec l'heuristique gloutonne qui nous sert de solution initiale.

			glouton			descent		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
ft06	6x6	55	2	62	12.7	27	60	9.1
ft10	10x10	930	1	1279	37.5	63	1084	16.6
ft20	20x5	1165	1	1714	47.1	34	1388	19.1
AVG	-	-	1.3	-	32.5	41.3	-	14.9

Figure 3: Test de la méthode de descente comparée avec la solution initiale gloutonne sur les instances ft06, ft10 et ft20

			glouton			descent		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
la01	10x5	666	3	816	22.5	37	747	12.2
la40	15x15	1222	6	1972	61.4	111	1768	44.7
AVG	-	-	4.5	-	41.9	74.0	-	28.4

Figure 4: Test de la méthode de descente comparée avec la solution initiale gloutonne sur les instances la01 et la40

Lorsque nous comparons les résultats obtenus à partir de la méthode de descente par rapport à la solution initiale (heuristique gloutonne) on remarque une amélioration des résulats par la réduction de l'écart avec la solution optimale. Cette amélioration montre que la façon dont on génère les voisins est une bonne méthodes.

Cependant la génération de voisin demande un temps de calcul supplémentaire non négligeable. En effet, on passe de quelques milliseconde à des centaines. Ce temps de calcul supplémentatire peut éventuellement être problématique dans certaine situation.

6 Méthodes Taboo

Nous avons finalement implémenté une heuristique taboo. Le principe de cette heuristique et la même que l'heuristique de descente mise à part que notre condition d'arrêt n'est plus le manque de meileur voisin mais un nombre d'itération maximum. En effet, même si le meilleur voisin trouvé n'améliore pas la solution courante on va quand même continuer à explorer dans sa direction. Ce principe va nous permettre de sortir des minimums locaux. Afin d'éviter un bouclage sur certains voisins nous avons mis en place un suivi des solutions taboos. Si un voisin a été exploré par le changement de deux tâches dans un blocs alors on empêche de réaliser le changement inverse pendant un certain nombre d'itération pour s'assurer de ne pas retourner dans notre minimum local.

Pour réaliser ce suivi on utilise une matrice dont les index sont les tâches correspondantes aux tâches qui sont échangées. Pour réaliser les tests nous avons dû fixer le nombre maximum d'itération et le nombre d'itération où l'on empêche les changements de tâches. Nous avons fixé le nombre d'itération à 3000 et le temps de changement à 5. Vous trouverez sur la Figure 5 les résultats obtenus en comparaison à l'heuristique de descente.

			descent			taboo		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
ft06	6x6	55	31	60	9.1	561	60	9.1
ft20	20x5	1165	29	1388	19.1	999	1388	19.1
la06	15x5	926	2	957	3.3	739	957	3.3
la40	15x15	1222	31	1768	44.7	999	1782	45.8
orb10	10x10	944	5	1499	58.8	999	1325	40.4
AVG	-	-	19.6	-	27.0	859.4	-	23.6

Figure 5: Test de la méthode taboo comparée avec la méthode de descente sur les instances ft06, ft20, la06, la10 et orb10

Nous pouvons tout d'abord noter que le résultat de la méthode taboo est soit égale soit meilleur à celui avec la méthode de descente. Ce résultat est logique car on part de la même solution initiale donc les deux algorithmes vont explorer le même premier minimum local. Lorsque la méthode taboo trouve un meilleur résultat cela signifie que le minimum local n'était pas le minimum global. Cependant, on ne trouve pas le meilleur résultat possible, cela est dû au nombre limite d'itération et de temps d'exécution.

En effet, le faite d'avoir changé notre condition d'arrêt par un nombre maximum d'itération fait exploser notre temps d'exécution. De plus, si l'on limite trop le nombre d'itération alors le résultat peut être moins bon que celui de la méthode de descente car l'on a pas le temps d'explorer le minimum local.

Si l'on désire trouver un minimum il vaut mieux utiliser la méthode de descente de part son temps d'exécution. Par contre, si l'on désire trouver un minimum global la méthode taboo est plus adaptée, mais il faudra alors faire des concessions sur le temps d'exécution de l'algorithme. De plus, afin de s'assurer de trouver un minimum global il faudrait jouer sur le temps où l'on empêche les changements entre deux tâches. Il faudrait l'augmenter afin de s'assurer que l'on s'éloigne bien du minimum local.

7 Glouton aléatoire et départ Multi Start

Dans le but de réaliser un algorithme de déscente multi start, nous avons d'abord ajouter de l'aléatoir dans notre heuristique gloutonne afin de pouvoir générer différente solution de départ. Notre heuristique fonctionne sur le même principe que l'heuristique gloutonne, la différence se situe sur le choix du critère. Avant nous choisissions un critère et nous l'utilisions pour chaque décision. Maintenant, lorsqu'une décision doit être prise, le critère est choisi au hasard parmis les quatres critères et ça à chaque fois. Vous trouverez sur la Figure 6 des tests de l'heuristique sur l'instance ft06.

```
                              randomGlouton
instance size  best      runtime makespan ecart
ft06     6x6     55            3       60   9.1
AVG      -        -          3.0        -   9.1
nce ft06
                              randomGlouton
instance size  best      runtime makespan ecart
ft06     6x6     55            3       72  30.9
AVG      -        -          3.0        -  30.9
nce ft06
                              randomGlouton
instance size  best      runtime makespan ecart
ft06     6x6     55            3       78  41.8
AVG      -        -          3.0        -  41.8
nce ft06
                              randomGlouton
instance size  best      runtime makespan ecart
ft06     6x6     55            2       58   5.5
AVG      -        -          2.0        -   5.5
```

Figure 6: Test de l'heuristique gloutonne avec hasard sur l'instance ft06

Pour coder notre algorithme de descente multi start nous démarrons dans la classe DescentMultiStart un nombre arbitraire de thread. Ces threads sont des instances de la méthode de descente précédemment développée mais dont la solution initiale est déterminée grâce à l'heuristaique gloutonne aléatoire. Ainsi chaque thread part d'une solution différente. Lorque que chaque thread à déterminer sa solution on récupère tout les résultats et nous gardons le meilleur. Vous trouverez sur la Figure 7 la méthode de descente multi start comparée à la méthode de descente classique avec huit threads.

			descent			descentMultiStart		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
ft06	6x6	55	30	60	9.1	27	58	5.5
ft10	10x10	930	40	1084	16.6	194	1098	18.1
ft20	20x5	1165	13	1388	19.1	38	1551	33.1
la06	15x5	926	3	957	3.3	52	926	0.0
la40	15x15	1222	48	1768	44.7	311	1642	34.4
orb10	10x10	944	6	1499	58.8	88	1385	46.7
AVG	-	-	23.3	-	25.3	118.3	-	23.0

Figure 7: Test de la méthode de descente multi start comparée avec la méthode de descente sur les instances ft06, ft20, la06, la40 et orb10

Nous remarquons que parfois la méthode de descente est meilleure que la méthode de descente pulti start. Cela est du au faite que la version multi start est dépendante de la solution initiale qui elle est choisie de façon plus ou moins aléatoire. Pour pallier à ce problème il faut alors augmenter le nombre de thread qu'utilise la méthode de descente multi start. En effet, plus l'on va augmenter le nombre de thread plus le résultat sera proche de la solution optimale. La méthode de descente pulti start donne donc de bon résultat à condition d'avoir assez de thread pour la dimension du problème.