Compte rendu de projet Métaheuristiques

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Hernandez Thibaut, Bamrung Premchanok et Delmon Thibault

Dans le cadre du cours *optimisation combinatoire*, il nous a été permis d'étudier ce que l'on appelle les Métaheuristiques. Aujourd'hui, on se tourne vers l'informatique pour essayer de résoudre des problèmes d'ordonnancement, qui sont regroupés dans ce que l'on appelle aussi le domaine de la recherche opérationnelle. Celui-ci regroupe toutes les méthodes qui permettent de répondre à des problèmes d'optimisation en termes de recherche de la meilleure méthode de procéder tant au niveau de la véracité d'une solution, qu'à la vitesse et donc le temps que met un algorithme pour l'atteindre. Dans ce projet, nous traitons un problème appelé Jobshop et traitons du calcul de dates optimales d'exécution pour des tâches données. Ce problème est l'un des plus connus car facile à comprendre et aussi adaptable à la difficulté de l'étude. Le cas d'étude est le suivant, on a *j* jobs contenant un nombre de tâches à exécuter ainsi qu'un nombre de ressources ou machines afin de traiter ces jobs. Le but, ici, est de trouver une solution d'ordonnancement qui permette de traiter toutes les tâches, de la première à la dernière, en un minimum de temps, on appelle cela aussi le makespan. Bien évidemment, nous nous heurtons à deux types de contraintes : ***temporelles*** (une tâche d'un job doit être terminée avant que la suivante du même job puisse commencer) et de ***ressources*** (une machine ne peut effectuer qu'une tâche à la fois). Ainsi, il nous a été donné en amont un code qui nous permet de prendre en main le sujet avec des solutions et des exemples. Il faut savoir que ce code est fourni en Java et que nous avons décidé de nous lancer dans ce langage et de laisser python de côté cette fois-ci. Le travail commence par la prise en main des codes existants et de comprendre leurs fonctionnements grâce aux tests de ces solutions. Ensuite, nous traiterons et créerons diverses méthodes afin de résoudre ces problèmes d'ordonnancement (gloutonne, descente, etc.) qui permettront de faire varier les choix d'explorations. Avant de conclure, nous effectuerons une étude comparative de ces méthodes.

Le code complet est récupérable en cliquant sur ce lien :

https://github.com/PremBamrung/metaheuristique.git

Table des matières

Table des figures 2
Section 1 : Modélisation du problème 3
Position du problème 3
Etude du code existant 4
Section 2 : Méthodes gloutonnes 6
Section 3 : Méthode de descente 8
Section 4 : Méthode tabou 9
Section 5 : Etude comparative 10
Les priorités entre glouton et descente 10
Meilleurs solveurs 12
Runtime 13
Conclusion 15

Table des figures

Figure 1 : Heuristiques gloutonnes pour les 4 types de priorités 7
Figure 2 : Heuristiques de descente pour les 4 types de priorités 9
Figure 3 : Comparaison meilleure descente avec méthode Tabou 10
Figure 4 : Comparaisons sur chaque priorité entre les gloutonnes et les descentes 11
Figure 5 : Toutes heuristiques gloutonnes et de descente 12
Figure 6 : Meilleurs solveurs de chaque méthode 12
Figure 7 : Ecart moyen aux résultats connus de chaque méthode 13
Figure 8 : Runtime moyens sur toutes les méthodes 14
Figure 9 : Runtimes des différentes méthodes 14

Avant toute chose, nous tenons à préciser que l'étude de comparaison des méthodes se fera en partie 5. Dans les trois types d'heuristiques suivants nous nous concentrons donc juste à savoir quelle technique ou méthode donne le meilleur résultat pour chacune.

Section 1 : Modélisation du problème

Position du problème

Voici le problème que nous étudions dans ce projet. Il s'agit du problème d'un Jobshop qui s'articule sur les hypothèses et méthodes suivantes :

- Un ensemble J de n Jobs (ou travaux) décomposé en une succession de tâches (ou activités ou opérations). Pour un job j, on note nj le nombre de tâches.
- D'un ensemble R de m ressources disjonctives pour réaliser ces tâches
- Toute opération (i, j) d'un job j nécessite une unique ressource donnée, notée k, pendant une durée p (i, j),
- Chaque ressource k est disponible en un seul exemplaire,
- Dans tout job j, l'opération (i, j) est successeur de l'opération (i − 1, j), ∀i ∈ [2, nj], les jobs peuvent débuter à la date 0.
- En d'autres termes :
    - Chaque tâche est attribuée à une machine et a un temps d'exécution propre,
    - Deux types de contraintes :
        - Temporelles : une tâche d'un job j doit être terminée avant que la suivante du même job puisse commencer,
        - Ressources : chaque machine a des tâches attribuées et ne peut en exécuter qu'une seule à la fois.

L'objectif est de minimiser la durée totale de l'ordonnancement aussi appelée makespan. La résolution de ce problème nécessite donc de déterminer la date de début de chaque opération. Un exemple concret et peut-être plus explicatif :

J_1	$r_1, 3$	$r_2, 3$	$r_3, 2$
J_2	$r_2, 2$	$r_1, 2$	$r_3, 4$

En clair, nous avons deux jobs $\boldsymbol{J_1}$ et $\boldsymbol{J_2}$ avec chacun trois tâches et trois ressources disponibles pour résoudre ce problème : $\boldsymbol{r_1}$, $\boldsymbol{r_2}$ *et* $\boldsymbol{r_3}$. Il faut savoir qu'il y a différentes façons de représenter des solutions afin qu'elles respectent les contraintes :

- Représentation détaillée d'une solution, qui consiste à stocker la date de début de chaque opération de chaque job.

num colonne	1	2	3
Job 1	0	3	6
Job 2	0	3	8

- Représentation par ordre de passage sur les ressources. Elle consiste à donner pour chaque machine l'ordre dans lequel réaliser les différentes opérations.

r_1	$(1,1)$	$(2,2)$
r_2	$(2,1)$	$(1,2)$
r_3	$(1,3)$	$(2,3)$

- Représentation par numéro de job. Elle se base sur un ordre entre les opérations des jobs. C'est-à-dire que l'on affiche l'ordre des jobs en relation avec l'ordre des tâches comme ici, les tâches 1 et 2 de $\boldsymbol{J}_1$ sont jouées puis la tache 1 du $\boldsymbol{J}_2$.

num job	1	1	2	2	1	2

Dans le cadre de ce projet, nous chercherons à représenter le problème sous la forme d'une représentation détaillée en passant par celle des ordres de passages. Ceci va permettre une meilleure représentation. Dans ce dernier, nous avons trois espaces de recherche, à savoir JobNumbers, Ressource Order et Schedule. Ces trois permettent d'étudier des instances fournies de tailles variables et qui dépendent donc de plusieurs paramètres comme le nombre de jobs et de tâches. Comme nous l'avions fait en cours, nous prenons l'exemple de l'instance ft06 qui contient 6 jobs et 6 machines pour étudier les temps d'exploration de tout l'espace de recherche, les limites et les avantages de chaque méthode. Pour cela, on étudie ces trois en supposant que la recherche et l'évaluation prend une nanoseconde.

- JobNumbers : taille de recherche $= \frac{(nb_{job} * nb_{machine})!}{(nb\_machine!)^{job}} = 2{,}67 * 10^{24}$, soit un temps $t = 2{,}67 * 10^{24} ns$ ce qui vaut 84 millions d'années.
- RessourceOrder : taille de recherche $= nb\_job!^{nb\_machine} = 1{,}39 * 10^{17}$, $t = 1{,}39 * 10^{17} ns$, soit environ 4 ans.
- Schedule : dans cette méthode il est nécessaire de connaitre la durée maximale, la durée du pire cas possible et dans notre cas on trouve un $D_{max} = 197$. Elle représente la somme de tous les temps de chaque tâche comme si elles étaient exécutées l'une après l'autre. On a donc : taille de recherche $= 3{,}99 * 10^{82}$, soit un temps $t = 3{,}99 * 10^{82} ns$, soit environ beaucoup trop d'années.

Cette rapide étude permet de montrer la difficulté d'obtenir une solution dans des temps raisonnables et qui peuvent varier selon la méthode utilisée. En effet, on voit que la représentation de l'espace de recherche a son importance et celle qui nous permet d'avoir un temps « moins pire », est la méthode de RessourceOrder. A cela s'ajoute le fait qu'explorer toutes les solutions n'est pas forcément un choix judicieux. C'est pour cela que dans la suite nous allons donc étudier des méthodes optimisées telles que des méthodes gloutonnes, de descente ou bien Tabou.

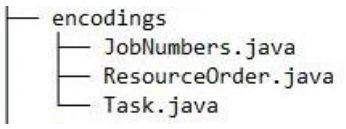
Etude du code existant

Dans ce projet, comme dit plus haut, il nous est fourni de base, un code principal qui permet d'étudier les différentes représentations, etc. Dans un premier temps il a fallu comprendre la

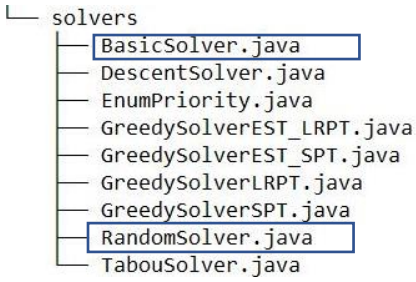
structure du code et le langage lui-même. En effet, nous sommes plus habitués à coder en python cependant dans ce projet nous trouvions cela plus intéressant de poursuivre en Java.

Voici la structure du code :

- Représenter des solutions d'instances grâce à la classe JobNumbers et RessourceOrder qui sont présentes dans un package appelé Encoding.

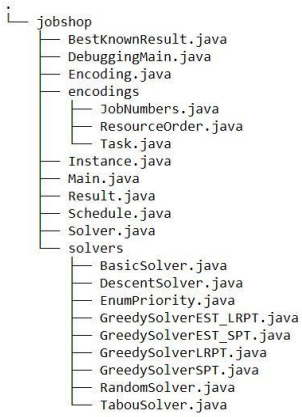


- Ensuite nous avons le package solvers qui contient à la fois les solvers de base et les solvers que nous avons créés au cours du projet :



  Ces deux classes encadrées étaient fournies et nous permettaient, sans modifications du code et à la réception de celui-ci, de pouvoir observer quelques résultats. Il s'agit ici de méthodes simples. A la suite de cela nous avons donc créé les trois autres classes qui correspondent aux méthodes optimisées et recherchées dans ce projet.

- On a donc créé plusieurs classes pour les heuristiques gloutonnes qui va permettre de gérer le choix entre des priorités telles que :
  - SPT (Shortest Processing Time) : donne la priorité à la tâche la plus courte,
  - LRPT (Longest Remaining Processing Time) : donne la priorité à la tâche appartenant au job ayant la plus grande durée,
  - EST_SPT, SPT + restriction aux tâches pouvant commencer au plus tôt,
  - EST_LRPT, LRPT + restriction aux tâches pouvant commencer au plus tôt.
- Et pour finir, il a fallu modifier le Main.java pour adapter le code à nos nouvelles méthodes.
- On obtient donc une structure de code comme suit :



Pour la suite de ce compte-rendu, nous allons traiter différentes instances sur différentes priorités. Le but de ceci est d'observer pour chaque instance le makespan (temps de traitement des tâches) selon la priorité appliquée.

Section 2 : Méthodes gloutonnes

On a pu voir dans la partie précédente que les temps d'exploration des solutions sont dépendants de l'espace de recherche et sont conséquents. Donc, plus l'instance est grande, plus le temps sera long. Pour cela, on va maintenant étudier différentes heuristiques qui vont nous permettre de réduire considérablement ce makespan. La première étudiée est l'heuristique gloutonne (tous les GreedySolver_.java).

Tout d'abord, il faut savoir qu'un algorithme est dit glouton si à chaque itération, la solution est choisie de manière à ce que ce soit un optimum local dans le but d'obtenir un optimum global. Cependant, il se peut que cet optimum global ne soit pas atteint, mais il donne une solution réalisable rapidement et c'est ce que l'on appelle une heuristique. Le but, ici, est d'obtenir un makespan possible et dans la limite du raisonnable pour différentes instances.

Avant de commencer, il est important de rappeler que des contraintes s'appliquent aux tâches et aux machines comme :

- Temporelles : une tâche d'un job j doit être terminée avant que la suivante du même job puisse commencer,
- Ressources : chaque machine a des tâches attribuées et ne peut en exécuter qu'une seule à la fois.

Pour cela, nous avons donc créé des priorités comme expliqué plus haut, à savoir :

```
public enum EnumPriority {
	SPT, //(Shortest Processing Time) : gives priority to the shortest task
	LRPT //(Longest Remaining Processing Time) : gives priority to the task be-
	longing to the job with the highest duration
	EST_SPT, //SPT + restriction to tasks that can start as soon as possible
	EST_LRPT, //LRPT + restriction to tasks that can start as soon as possible
}
```

Ainsi, nous allons donc tester plusieurs instances sur ces différentes règles de priorités et comparer les makespan afin d'observer laquelle des quatre propose la meilleure solution.

Le choix de la priorité est donné par le choix de la classe 'Greedy' associée à cette dernière. On instancie toutes les données du problème comme l'instance choisie, le nombre de jobs et de tâches et ensuite, on fait appel à une de ces priorités. Il est important de bien comprendre l'algorithme, car celui-ci est exécuté jusqu'à ce qu'il ne reste plus aucune tâche à traiter et c'est pour cela que nous faisons une boucle *while* qui nous permettra de retourner une solution.

Par soucis de visibilité, nous ne présentons pas le code sur ce rapport, mais il est bien évidemment disponible avec le lien en page 1.

Ainsi, après avoir codé cette méthode nous avons procédé à des tests sur différentes instances comme : *ft06* (6x6), *la01* (10x5), etc. Pour résumer, nous aurons donc comme résultat retourné un tableau constitué de chaque instance pour les lignes et pour les colonnes, le runtime (temps de calcul) et le makespan (meilleur temps d'exécution des tâches). Voici un exemple (il ne s'agit pas du vrai tableau, mais d'une ébauche plus visible et compréhensible) :

Instance			SPT		LRPT		EST_SPT		EST_LRPT	
name	size	best	runtime	makespan	runtime	makespan	runtime	makespan	runtime	makespan
ft06	6x6	55	0	108	0	97	0	77	0	63

Une fois la méthode gloutonne créée, on peut lancer les algorithmes et observer les résultats suivants :

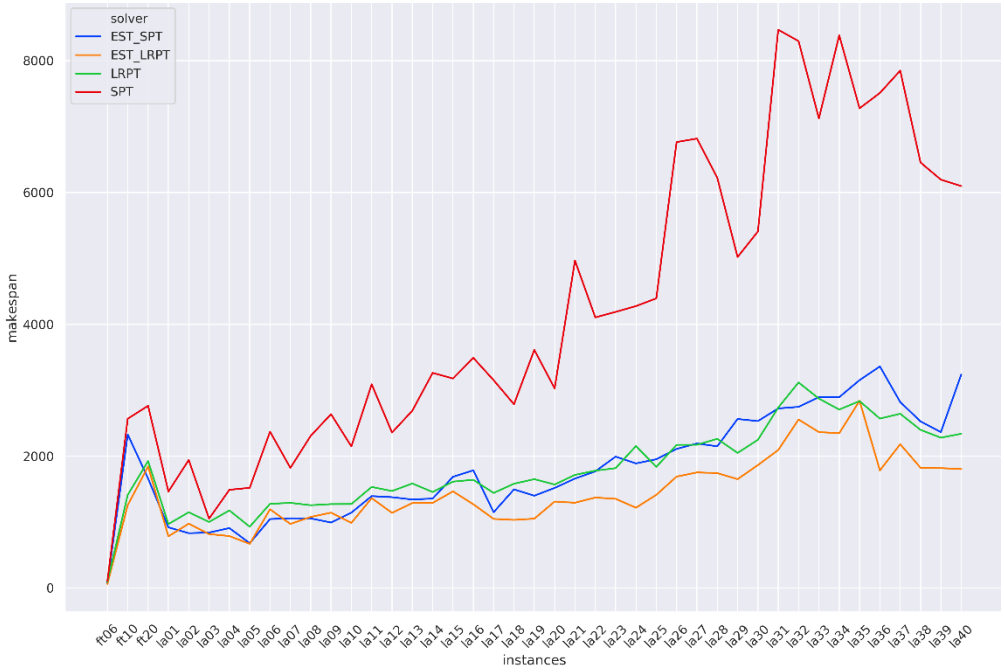


Figure 1 : Heuristiques gloutonnes pour les 4 types de priorités

Le graphe ci-dessus nous montre les différents makespan de chaque priorité en fonction de la taille de l'instance, classées par ordre croissant. La superposition des courbes permet de visualiser l'impact de la priorité choisie.

On peut voir que lorsque les instances commencent à devenir plus grandes, on a une différence de temps entre la méthode SPT et les méthodes LRPT, EST_SPT et EST_LRPT, le makespan va au-delà du double. La méthode basée sur la priorité SPT diverge très vite et atteint des makespan très éloignés des trois autres.

Si l'on rentre plus dans les détails, on voit que la priorité EST_LRPT est la meilleure quoiqu'il arrive, et ce pour tous les types d'instances. Ainsi, le choix serait de partir sur une méthode gloutonne basée sur la priorisation EST_LRPT. On donne la priorité à la tâche appartenant au job avec la durée la plus élevée plus une restriction qui impose des restrictions à la tâche qui peut démarrer le plus vite possible.

Il faut aussi observer que pour chaque priorité, le makespan est loin d'être aussi bon que les résultats officiels trouvés sur ces méthodes. Ce qui montre que nous sommes sur la bonne voie, mais il reste des améliorations possibles.

Bien sûr, dans le domaine de la recherche opérationnelle, les méthodes gloutonnes ne sont pas considérées comme les plus efficaces en termes de précision. Il s'agit d'une méthode qui est discutable car on ne se concentre que sur un calcul à l'instant t. C'est-à-dire que l'on n'anticipe pas et que l'on peut rester bloqué sur un minimum local. Donc, nous avons une solution possible, dont le makespan a été optimisé, mais on ne renvoie pas forcément le meilleur. Ceci est l'un des inconvénients de cette méthode. Mais cela reste quand même raisonnable comme cas d'étude.

Section 3 : Méthode de descente

On va s'intéresser maintenant à la méthode de descente et voisinage. Tout d'abord, une définition théorique qui dit que l'idée générale d'une méthode de descente est de toujours prendre dans un voisinage une solution meilleure que la solution courante. Pour cela, il existe différentes pistes d'études et d'explorations comme l'exploration aléatoire, l'exploration systématique ou même l'exploration exhaustive pour déterminer le meilleur voisin.

Dans notre projet, la méthode que nous avons implémentée ne s'occupe pas de cette partie exploration des voisins, car nous avons choisi de dire qu'elle a une meilleure solution à chaque étape ou bien elle s'arrête. On peut donc dire qu'elle s'arrête sur un minimum local ou bien après dépassement d'un paramètre identifié comme étant un temps limite. Ainsi, on considère que la solution après une itération est soit meilleure soit égale à la précédente.

Il est important de relever la notion de chemin critique, puisque nous pouvons en extraire la liste des blocs dont il est composé : un bloc contient au minimum deux tâches consécutives utilisant la même ressource/machine. Pour une solution donnée, on peut trouver le chemin des tâches consécutives tel que la somme des temps d'exécution de ces tâches soit maximale. A partir de là, un bloc peut alors subir un swap, c'est à dire que, deux tâches au sein d'un même bloc peuvent être échangées. Ainsi, pour une solution donnée, on obtient donc une liste de swap acceptables pouvant lui être appliquée.

Nous nous sommes aussi penchés sur le type de priorité que nous allons implanter dans cet algorithme de descente. Dans cette partie, nous avons décidé de tester notre algorithme de

descente avec les 4 types de priorités possibles. On aurait pu se dire, étant donné qu'EST_LRPT marchait mieux chez le glouton, on n'effectue que cette priorité sur la descente mais d'un point de vue validation nous avons fait sur toutes.

Ainsi, grâce au graphe suivant, nous avons affiché le makespan pour toutes les priorités possibles pour les instances *ft** et *la** classées par ordre croissant en termes d'ordre :

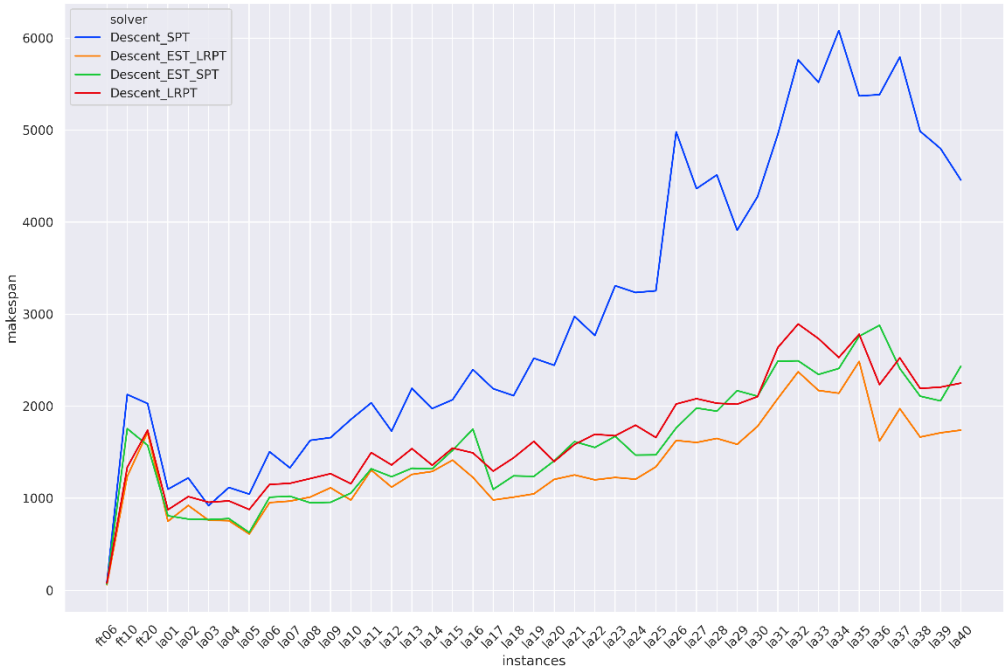


Figure 2 : Heuristiques de descente pour les 4 types de priorités

On peut faire quelques remarques sur ce graphe qui ressemble beaucoup à celui des heuristiques gloutonnes :

- Pour de petites instances, le makespan reste raisonnable et surtout on n'observe pas trop de différences entre les priorités SPT, LRPT, EST_SPT et EST_LRPT.
- Plus on augmente la taille des instances et plus on voit que la priorisation des tâches est importante. En effet, on voit que, comme pour la gloutonne, l'EST_LRPT reste la meilleure priorité.

Pour la suite de cette étude on va donc appliquer un algorithme basé sur la méthode de descente avec une priorité de type EST_LRPT.

Section 4 : Méthode tabou

Pour finir, nous allons maintenant nous intéresser à la méthode appelée Tabou. C'est une technique de recherche de la meilleure solution et donc d'optimisation, basée sur l'exploration de voisinage. Ce qui la différencie des précédentes c'est que celle-ci va nous permettre de sortir des optima locaux. Pour se faire, elle va accepter des solutions qui auraient été considérées comme non-améliorantes dans les précédents algorithmes mais qui peuvent aider à obtenir le minimum global. Une particularité qui est bien utile dans cette méthode est

qu'elle va garder en mémoire les solutions déjà vues pour éviter de repasser dessus afin de ne pas rester bloquée.

Contrairement aux méthodes étudiées précédemment nous avons choisi de n'étudier que la priorité de type EST_LRPT. En effet, on a pu se rendre compte que pour quasiment toutes les instances, la priorité qui donnait le meilleur makespan était celle-ci. Ainsi, nous avons comparé cette méthode Tabou avec la méthode de descente sous la priorité EST_LRPT. En voici le graphe résultant :

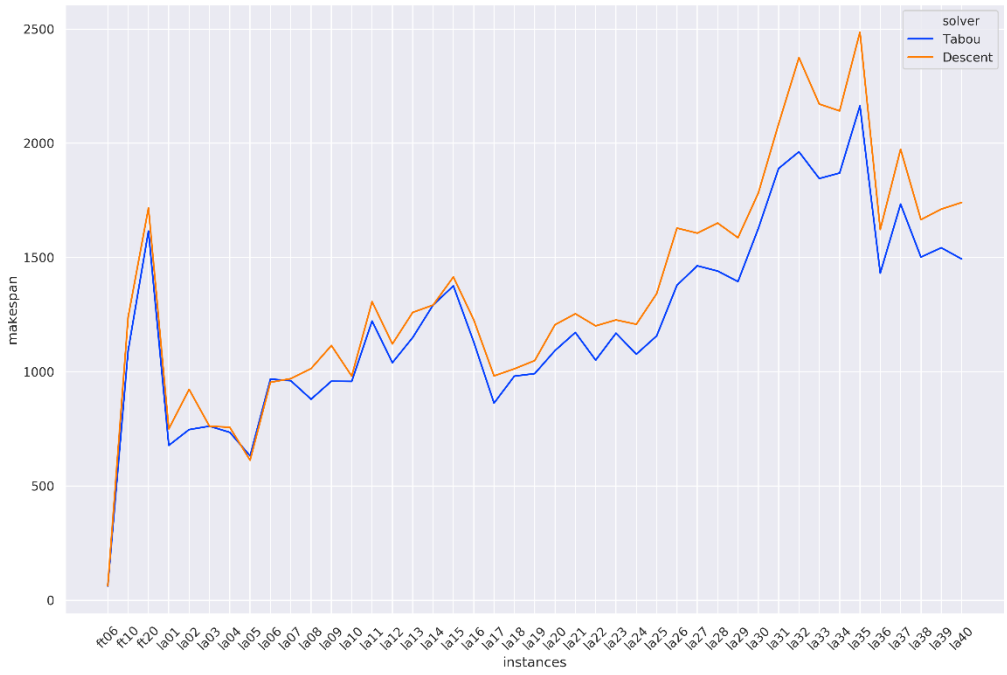


Figure 3 : Comparaison meilleure descente avec méthode Tabou

Ce graphique nous permet de voir qu'effectivement, la méthode Tabou permet une amélioration par rapport à la meilleure heuristique de descente. Cette différence est quand même légère pour les petites instances mais visibles plus la taille augmente. On peut estimer que la méthode tabou, sous la priorisation EST_LRPT, reste la plus performante.

Section 5 : Etude comparative

Maintenant, nous avons étudié chaque méthode en détail, la suite logique est la comparaison des méthodes Gloutonnes, de Descente et Tabou.

Les priorités entre glouton et descente

Dans un premier temps, nous allons comparer chaque type de priorité entre la méthode gloutonne et celle de la descente pour voir si cette dernière améliore en tout point l'algorithme glouton.

En effet, nous savons que pour chaque méthode, le meilleur makespan obtenu pour chaque instance est obtenu grâce à la priorité EST_LRPT. Cependant, nous voulons voir s'il y a une différence entre les deux méthodes.

Pour cela, nous avons tracé chaque priorité sur un graphe avec à la fois la méthode gloutonne et la méthode de descente associée.

Figure 4 : Comparaisons sur chaque priorité entre les gloutonnes et les descentes

On peut voir grâce à ces quatres graphiques que la méthode de descente reste la plus performante quelle que soit la priorisation. En effet, la descente (en orange) retourne de meilleurs makespan pour tous les types de priorités. Cependant on peut faire les mêmes remarques que précédemment, plus la taille des instances augmente et plus la séparation entre les deux méthodes est visible.

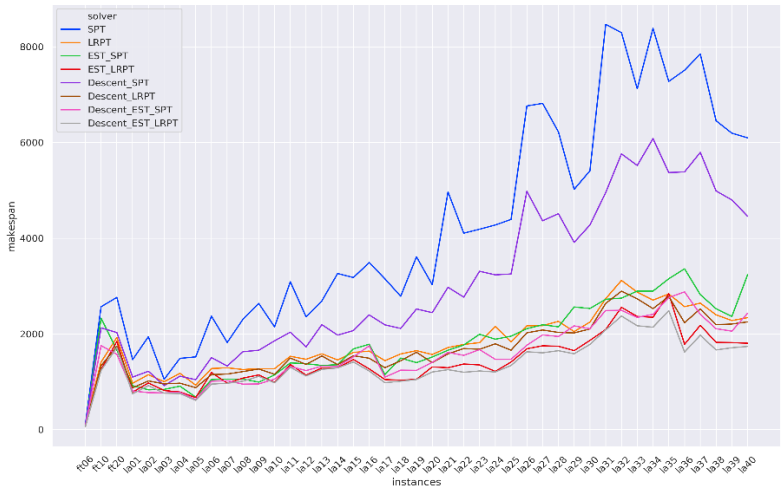


Figure 5 : Toutes heuristiques gloutonnes et de descente

On peut observer avec ce graphe, que la meilleure priorisation pour chaque méthode est la EST_LRPT. De plus, la différence de makespan entre la meilleure de chaque méthode (gloutonne et descente), est faible et s'observe surtout pour des grandes instances.

Ainsi, on peut dire que :

$$H\_descente_{EST_{LRPT}} \; est \; plus \; performante \; que \; H\_gloutonne_{EST_{LRPT}}$$

Meilleurs solveurs

Dans un second temps nous comparons les meilleurs solvers de chaque méthode pour savoir qui est le meilleur et si notre algorithme le plus performant se rapproche des meilleurs makespan «officiels».

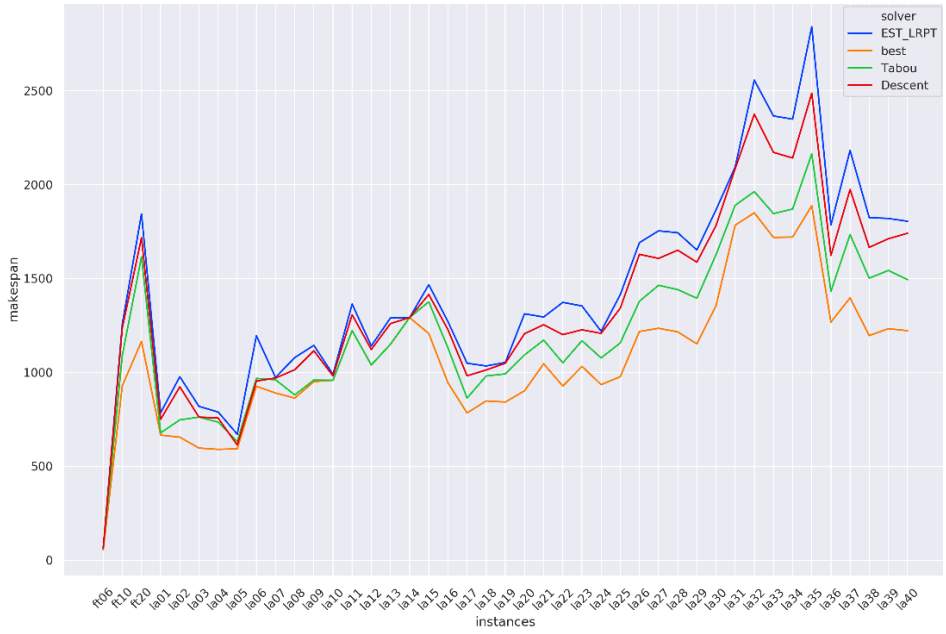


Figure 6 : Meilleurs solveurs de chaque méthode

Sur ce graphe, nous avons donc tracé les méthodes gloutonnes, de descente et tabou sous la priorisation EST_LRPT (meilleure pour chaque) ainsi que les résultats « officiels » (représenté dur le graphe par *best*) des meilleurs makespan trouvés. Ceci est intéressant, car cela nous prouve donc que la méthode tabou est quasiment toujours la meilleure méthode, du moins celle qui retourne le meilleur makespan. En plus de cela, il est important de relever que pour quelques instances, on arrive même à égaler le meilleur makespan. Ainsi, la meilleure méthode en termes de makespan est donc la méthode tabou sous la priorisation EST_LRPT. Dans laquelle on donne la priorité à la tâche appartenant au job avec la durée la plus élevée plus des restrictions à la tâche qui peut démarrer le plus vite possible. Pour finir, ce que l'on peut relever est le fait que le solver random soit en deuxième position.

Sur le graphe suivant, on a représenté l'écart moyen de makespan de chaque méthode avec les résultats « officiels ». On voit très clairement que la méthode tabou reste la meilleure et est très proche.

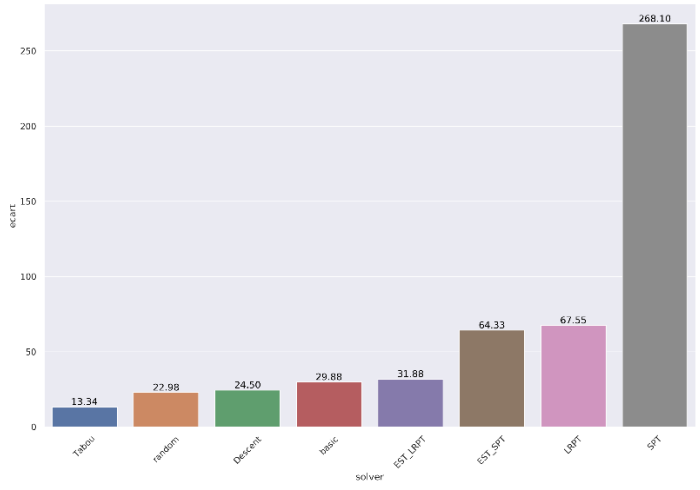


Figure 7 : Ecart moyen aux résultats connus de chaque méthode

Runtime

Les algorithmes ont été exécutés sur une machine avec les caractéristiques suivantes :

i7 4770k 3.9GHz 8 threads 16Go Ram

Et pour finir nous parlerons d'une notion importante, le runtime qui n'est pas à négliger. En effet, il est indispensable de faire attention à cette notion le rapport temps/performance est à prendre en compte. Pour expliciter cela, nous avons calculé les runtime moyen pour chaque méthode :

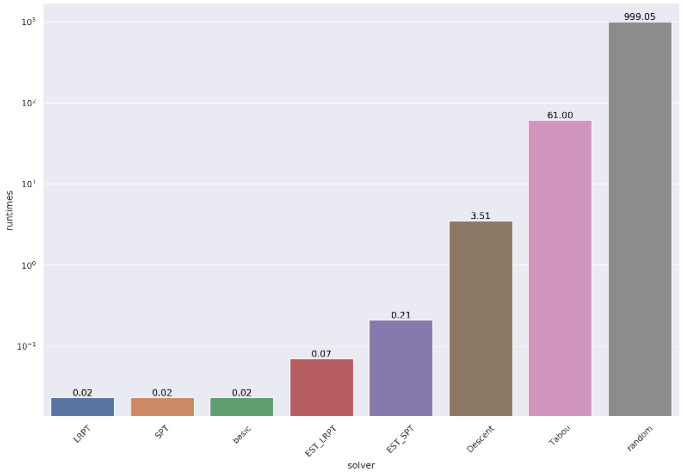


Figure 8 : Runtime moyens sur toutes les méthodes

On voit que la méthode Tabou prend énormément de temps en comparaison avec la méthode gloutonne LRPT, on est sur un rapport de plus de 3 000. Également, pour la méthode de descente avec un temps de calcul 175 fois supérieur à la méthode gloutonne LRPT. On observe donc que les algorithmes gloutons ne sont pas forcément les plus efficaces mais restent néanmoins les plus rapides. Ainsi, tout ceci soulève la question souvent posée, est-il plus avantageux d'avoir une méthode moins précise mais plus rapide ou bien plus précise et plus lente ? Bien sûr, tout ceci est discutable en fonction du cas d'étude. En effet, sur des instances simples, on peut se permettre d'effectuer une méthode Tabou, mais qu'en est-il sur les grandes instances ?

Voici un dernier graphe qui montre l'évolution des runtimes des méthodes en fonction de la taille des instances. On arrive bien à la conclusion que la méthode tabou prend de plus en plus de temps quand la taille augmente. D'où cette importance du rapport temps/performance.

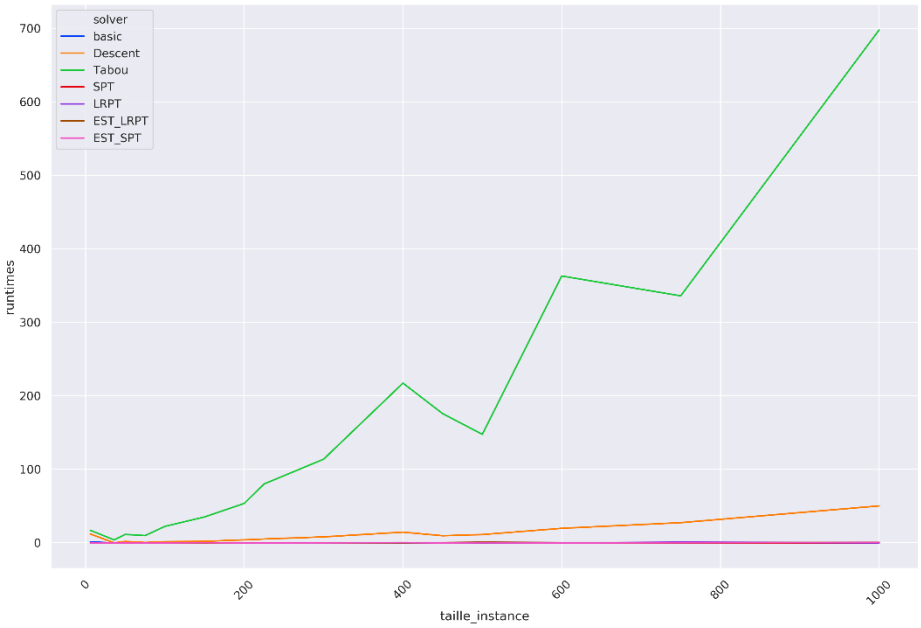


Figure 9 : Runtimes des différentes méthodes

Conclusion

Pour conclure, ce projet a été pour nous l'opportunité de mettre en pratique les connaissances abordées dans le cadre du cours de métaheuristique. De même, cela nous a permis de prendre ou reprendre le codage en Java. Au-delà des compétences en matière de code, il aura fallu bien comprendre toutes les techniques utilisées lors de ce projet comme la théorie qui se cache derrière une méthode dite Tabou. De plus, les cours que nous avons pu avoir, nous ont permis de bien appréhender cette thématique et nous avons trouvé cela intéressant de comparer différentes techniques sur des instances dont le meilleur makespan n'est pas forcément connu. Ce cas d'étude du Jobshop est à la fois un cas pratique et perfectible, car nous avons découvert ces méthodes avec des cas simples. Cependant, on a pu voir que pour certaines instances, le makespan optimal n'a pas été encore trouvé.

En termes de connaissances, nous avons pu traiter trois grandes méthodes connues aujourd'hui, à savoir les heuristiques gloutonnes, de descente et Tabou. Il a été intéressant de les étudier dans cet ordre-là car on a pu voir l'évolution du code avec une méthode gloutonne que l'on peut qualifier de brutale et non-optimale, afin d'arriver à une méthode tabou qui se rapproche et même parfois égale le meilleur makespan. Pour finir, nous pouvons dire qu'en terme d'optimalité, la meilleure méthode est celle dite tabou, mais il faut faire attention au temps de calcul. En effet, en termes de temps, la méthode tabou est lente. Vient la question du rapport temps/performance.

Au niveau du code, nous avons choisi Java, car cela nous permettait d'éviter de recoder en python toutes les classes. Cependant, il a fallu se réhabituer à l'obstacle du code java que l'on n'a pas l'habitude de traiter. Mais dans le fond, cela a été plutôt une adaptation rapide.

En somme, ce projet a été pour nous bénéfique et nous a permis d'avoir une très bonne approche de la notion de métaheuristique.