INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Département de Génie Électrique & Informatique

Rapport Metaheuristiques

Méthodes Approchées pour la Résolution de Problèmes d'Ordonnancement

22 mai 2020

Rédacteur :

SAHRAOUI Hichem

Encadrants :

BIT-MONNOT Arthur
HUGUET Marie-Jo

Promo 54 : 2019/2020

4IR

Table des matières

Introduction

Ce rapport présente mes travaux dans le cadre des travaux pratiques de Métaheuristiques. Ces travaux pratiques, assimilable à un projet, entièrement en distanciel, avaient pour ambition de nous faire découvrir différentes méthodes approchées pour la résolution de problèmes d'ordonnancement dans la continuité des cours magistraux de Métaheuristiques, dans l'Unité de Formation Systèmes Intelligents en quatrième année Informatique et Réseaux.

C'est ainsi qu'après avoir réalisé les exercices préparatoires au TP, il nous a été demandé de développer des morceaux de codes, des fonctions, parfois des solveurs, appliqués au problème de JobShop. Ce rapport explicite, toujours dans le cadre du problème de Jobshop, les heuristiques gloutonnes, comment je les ai implémentées, mais aussi plusieurs méthodes, en particulier descente & Taboo

Il est également explicité dans ce rapport les choix d'implémentations, les tests et les résultats obtenus, ainsi que les difficultés rencontrées et comment ils ont été résolues. Enfin figure dans un dernier chapitre une analyse des différentes méthodes.

L'ensemble du code source de ce projet est accessible dans un répertoire *git* :
https://github.com/hsn31/Metaheuristique.

1 Contexte

Dans ce chapitre est présenté le contexte général du projet, l'architecture illustré via un diagramme UML, mais aussi une ou deux difficultés rencontrées au cours de ces travaux pratiques.

1.1 Prise en main du code existant

Il nous a été fourni un dépôt *git*, composé de 3 packages :

- Le package `jobshop` contient notamment le main, point d'entrée pour réaliser les tests de performance de méthodes heuristiques.
- Le package `jobshop.encodings` qui contient plusieurs classes permettant la représentation d'une solution au JobShop, notamment (`JobNumbers` ou encore `ResourceOrder`)
- Le package `jobshop.solvers` qui contient des solveurs basés sur une représentation par numéro de job.C'est également dans ce package que j'ai ajouté au fur et à mesure les solveurs demandés (`Greedy`, `Descent` et `Taboo`)

Le dépôt contient également un package dédié au test, mais je l'ai peu utilisé préférant, par manque de temps surtout, des tests via des `println` et des comparaisons avec les fichiers `result.txt` fournis par les professeurs.

Si on entre un peu plus dans les détails, en faisant une analyse par classes, on remarque :

- Nous avons une classe abstraite `Encoding` qui, via ces sous classes nous permet de créer les différentes représentations du problème.
- Nous avons également une interface `Solver` qui permet de spécifier le comportement que les classes du package `jobshop.solvers` doivent implémenter.
- On retrouve en bref une classe `Instance` qui contient la représentation d'un problème.
- Mais aussi un classe `Schedule` qui contient la représentation directe, associant à chaque tâche une date de début.

Ainsi, ce projet contenait un nombre important de classes, regroupées en package, un des défis fut de comprendre leur fonctions, les liens, relations et dépendances entre ces classes et surtout comment nous devions nous en servir pour coder les fonctions et les solveurs demandés. Cela a incontestablement demandé un temps important, pouvant expliquer un peu le retard que j'ai pu prendre au début du projet.

Afin de mieux visualiser l'architecture du projet, j'ai inclu dans ce rapport un diagramme UML.

1.2 Diagramme UML

Voici un diagramme de classe du projet (figure 1), dans lequel est représenté l'ensemble des relations entre les classes rapidement explicité dans la sous partie précédente.

1.3 Le travail à réaliser

Le travail à réaliser fut double. Implémenter des méthodes principalement du package `jobshop.encodings`, puis ajouter des solvers au package `jobshop.solvers`.

Nous devions dans un premier temps implémenter la méthode `toString()`de la classe `Schedule` pour afficher les dates de début de chaque tâche dans un schedule.

Par ailleurs, il nous avait été demandé de créer une classe contenant une représentation sous forme de matrice où chaque ligne correspond à une machine, et sur cette ligne se trouvent les tâches qui

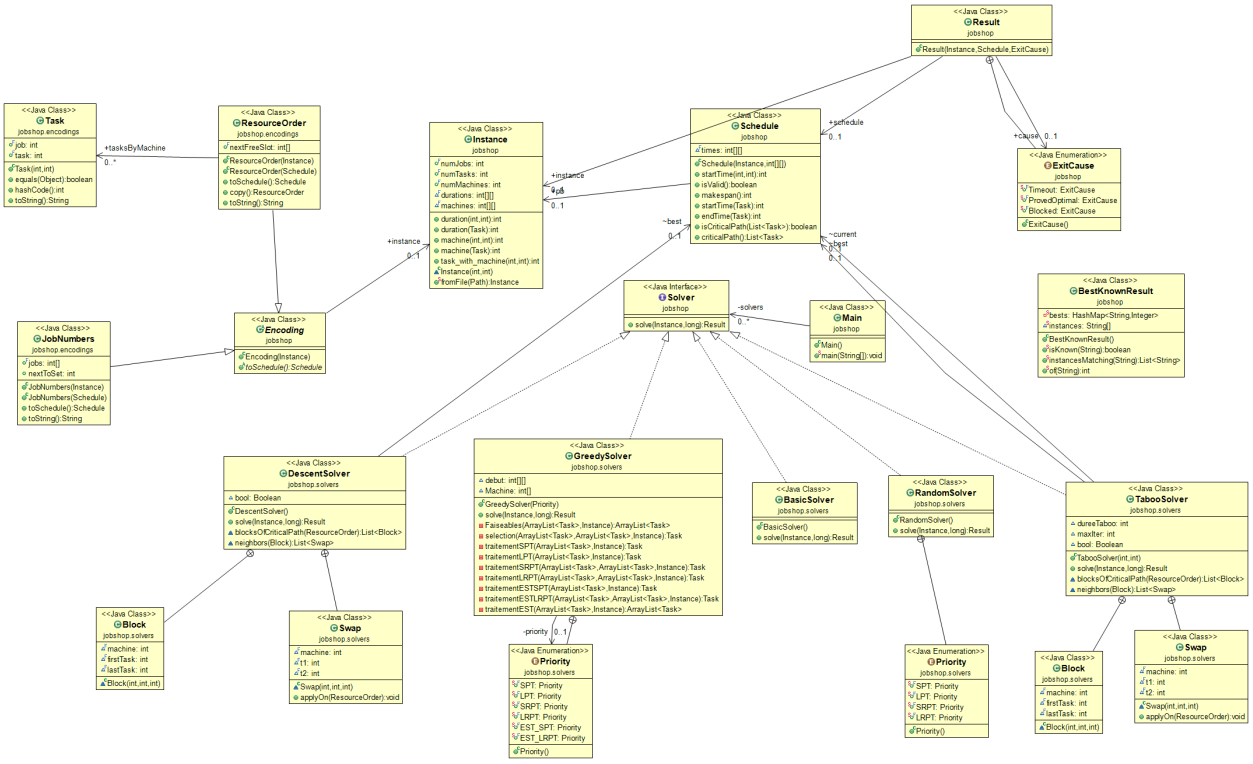

FIGURE 1 – Diagramme de classe du projet

s'exécutent dessus dans leur ordre de passage. Nous avons dans cette classe, implémenté la méthode `toSchedule()` qui permet d'extraire une représentation directe.

Nous avons ensuite créé un nouveau solveur (`GreedySolver`) implémentant une recherche gloutonne basée sur `ResourceOrder` puis une amélioration de ce solveur.

Nous nous sommes ensuite concentré sur `DescentSolver`, en écrivant une méthode `blocksOfCriticalPath` pour extraire la liste des blocs d'un chemin critique, une méthode `neighbors` pour générer le voisinage d'un bloc, puis `applyOn` de la classe Swap, avant de coder la méthode `solve`.

Enfin, nous avons ajouter un dernier solver, `TabooSolver` en s'inspirant des autres solveurs déjà réalisés.

1.4 Les difficultés

La principale difficulté fut d'ordre matériel, l'utilisant et la manipulation de Eclipse. J'étais sous Windows mais mon ordinateur n'était pas assez puissant pour faire tourner pendant l'ensemble de la durée des TP une machine virtuelle sous Linux. Par ailleurs, ma connexion internet, très faible depuis le domicile de mes parents, a rendu impossible l'utilisation du site montp.insa-toulouse.fr, permettant d'accèder à nos sessions INSA.

Il a donc fallu gérer sous windows les aléas liés à Eclipse, à l'installation de Gradle etc. Voici un exemple d'erreurs récurrentes levées par Eclipse. J'ai réussi à résoudre les erreurs via plusieurs camarades de promo qui avaient connus les mêmes difficultés avec Eclipse/Windows.

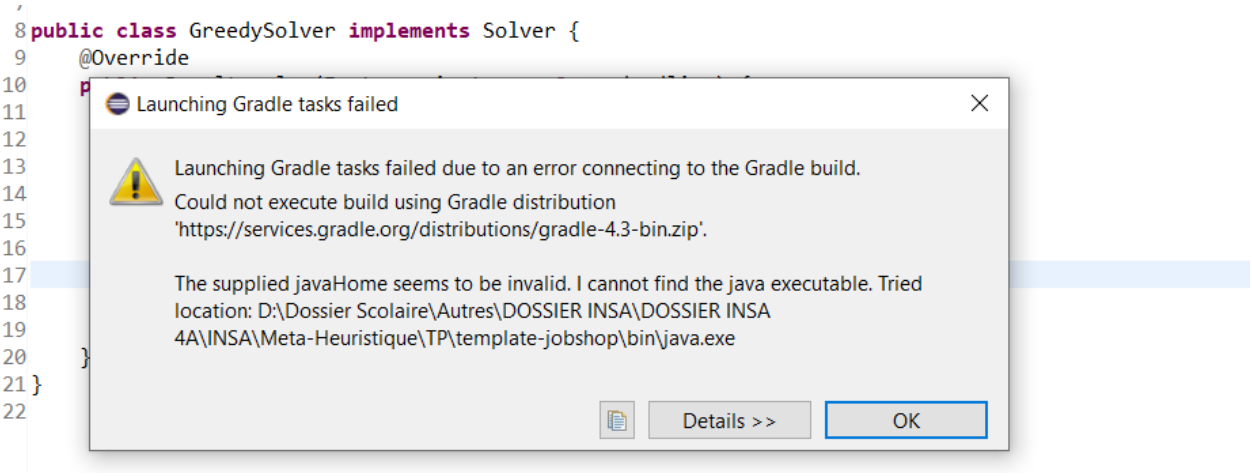

FIGURE 2 – Exemple de difficultés rencontrées avec Eclipse

La seconde principale difficulté fut de bien s'organiser en télétravail. Les cours en distanciel sont beaucoup plus chronophages et épuisants que les cours en présentiel. Il a donc fallu gérer cette situation inédite, apprendre à utiliser de nouveaux moyens, suivre des CM sous Discord etc. Un semestre en distantiel fut un défi, que je crois relevé.

2 Représentation de solutions et Espace de recherche

Le JobShop est un problème d'ordonnancement dans lequel on dispose d'un ensemble de n Jobs décomposé en une succession de tâches et d'un ensemble de m ressources (ou machines) disjonctives, c'est à dire ne pouvant exécuter qu'une tache à la fois, pour réaliser ces tâches. L'objectif est de minimiser la durée totale de l'ordonnancement.

Nous allons présenter les différentes représentations de solutions, et comparer en fonction de l'instance ft06.

```
#+++++++++++++++++++++++++++++
# instance ft06
#+++++++++++++++++++++++++++++
# Fisher and Thompson 6x6 instance, alternate name (mt06)
6 6
2  1  0  3  1  6  3  7  5  3  4  6
1  8  2  5  4 10  5 10  0 10  3  4
2  5  3  4  5  8  0  9  1  1  4  7
1  5  0  5  2  5  3  3  4  8  5  9
2  9  1  3  4  5  5  4  0  3  3  1
1  3  3  3  5  9  0 10  4  4  2  1
```

2.1 Représentation par numéro de job

Nous avons commencé les TP par une représentation par numéro de job, notamment via le solveur Basic. Cette représentation se base sur un ordre entre les opérations des jobs. Dans la méthode `DebuggingMain.main()`, nous avons une solution en représentation par numéro de jobs suivante :

```
[0 1 1 0 0 1]
```

Il nous est donné 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).

J'ai donc calculé à la main les dates de début de chaque tâche.

On obtient les résultats suivants :

Pour le Job 0 :
La tache 0 commence au temps 0
La tache 1 commence au temps 3
La tache 2 commence au temps 6.

Concernant le Job 1 :
La tache 0 commence au temps 0
La tache 1 commence au temps 3
La tache 2 commence au temps 8.

Puis j'ai confirmé en implémentant la méthode `toString()` de la classe `Schedule`. Cela fonctionne.

Concernant l'espace de recherche, la taille de l'espace de recherche dépend de la représentation choisie. Dans notre cas, avec la représentation par numéro de job (NumJobs), cette taille correspond au nombre de permutations du numéro de jobs avec répétition. Elle est égale à (nm) !/(m !)n̂.

Le nombre de solutions de l'espace de recherche associé à cette représentation pour l'instance ft06 est donc de : 2,67x10$\hat{}$24 solutions.

En supposant que la génération et l'évaluation d'une solution prend une nano-seconde (pour chacune des représentations), le temps nécessaire pour explorer l'espace de recherche associé sera donc de : 2,6x10$\hat{}$15 secondes soit 7,4x10$\hat{}$11 heures.

2.2 Représentation par ordre de passage

Nous nous intéressons maintenant à une représentation par ordre de passage sur les ressources. Il s'agit d'une représentation plus synthétique consistant à donner pour chaque machine l'ordre dans lequel réaliser les différentes opérations.

Avec la représentation par ordre de passage sur les ressources, cette taille correspond à toutes les permutations de tâches sur les ressources. Elle est égale à (n !)$\hat{}$m où n est le nombre de jobs et m le nombre de machines.

Le nombre de solutions de l'espace de recherche associé à cette représentation pour l'instance ft06 est donc de : 1,4x10$\hat{}$24 solutions.

En supposant que la génération et l'évaluation d'une solution prend une nano-seconde (pour chacune des représentations), le temps nécessaire pour explorer l'espace de recherche associé sera donc de : 1,4x10$\hat{}$8 secondes soit 39 000 heures.

2.3 Représentation par date de début de chaque tâche

Nous nous intéressons maintenant à une représentation par date de début de chaque tâche. Avec cette représentation, la taille dépend des valeurs possibles des dates de début et du nombre de tâches. Elle est égale à (Dmax)$\hat{}$(nm) où Dmax représente la durée maximale d'un ordonnancement.

J'ai pris Dmax comme la somme des durée de toutes les tâches. Ainsi Dmax= 197

Le nombre de solutions de l'espace de recherche associé à cette représentation pour l'instance ft06 est donc de : 3.9x10$\hat{}$82 solutions.

En supposant que la génération et l'évaluation d'une solution prend une nano-seconde (pour chacune des représentations), le temps nécessaire pour explorer l'espace de recherche associé sera donc de : 3,9x10$\hat{}$73 secondes soit 1.1x10$\hat{}$70 heures.

2.4 Analyse et Interprétation

On voit bien que les méthodes exhaustives, consistant à explorer la totalité des solutions ne sont pas efficaces pour ce problème en particulier, car les temps d'exécution sont beaucoup trop importants.

3 Heuristiques Gloutonnes

Ce chapitre aborde la question des Heuristiques Gloutonnes, notamment le principe général des méthodes implémentées, le tableau des résultats obtenus ainsi qu'une rapide analyse des résultats.

3.1 Principe général des méthodes implémentées

Voici un pseudo code de l'algorithme implémenté pour la méthode public `solve`, dans la classe `GreedySolver`. Ce pseudo code n'est pas exhaustif, et ne tente de refléter que la philosophie générale de l'algorithme.

Algorithm 1: méthode public solve - Heuristique Gloutonne

```
Data: Instance, Deadline
Result: Result
sol ← ResourceOrder(Instance);
taches_faiseables ← [ ];
taches_realisees ← [ ];
for tache ∈ instance.numJobs do
    add(taches_faiseables, tache);
end
while taches_faiseables != null do
    tasksByMachine ← 0;
    task ← selection(taches_faiseables, taches_realisees, Instance);
    numMachine ← instance.machine(task.job,task.task) ;
    jobnum ← 0;
    while sol.tasksByMachine [numMachine ] [jobnum ] != null do
        jobnum ++ ;
    end
    add(sol, task);
    add(taches_realisees, task);
end
result ← sol.toSchedule() ;
```

Afin d'implémenter cette méthode gloutonne, j'ai suivi les indications du sujet étape par étape, à savoir :

- Initialisation : J'ai essayé de déterminer l'ensemble des tâches réalisables (en utilisant la boucle for).
- Boucle : tant qu'il y a des tâches réalisables :
    1. On choisi une tâche dans cet ensemble en fonction des règles de priorité
    2. On détermine le numéro de la machine puis on place cette tâche sur la machine demandée
    3. On met à jour l'ensemble des tâches réalisables et réalisées.

Il existe plusieurs priorités différentes permettant de choisir la prochaine tâche à exécuter parmi toutes les tâches réalisables.

- SPT (Shortest Processing Time) : donne priorité à la tâche la plus courte ;
- LPT (Longest Processing Time) : donne priorité à la tâche la plus longue ;
- SRPT (Shortest Remaining Processing Time) : donne la priorité à la tâche appartenant au job ayant la plus petite durée restante ;

- LRPT (Longest Remaining Processing Time) : donne la priorité à la tâche appartenant au job ayant la plus grande durée

Nous avons ensuite chercher à améliorer ces méthodes gloutonnes en limitant le choix de la prochaine tâche à celles pouvant commencer au plus tôt. Nous avons ainsi le traitement EST (Earliest Start Time), appliqué avant les règles de priorité. Il s'agit des règles `EST_SPT` et `EST_LRPT`.

Concrètement, j'ai implémenté une méthode `private ArrayList<Task> Faiseables` ayant pour but de déterminer si une tâche est réalisable (appelé faiseable dans mon code pour éviter de confondre avec réalisés) si tous ses prédécesseurs ont été traités. Puis quatre méthodes traitementXX visant à appliquer la priorité et renvoyer la tache. C'est la méthode selection qui selectionne et appel le bon traitement via un case.

Concernant les `EST`, par efficacité, j'ai développé une méthode `TraitementEST`, applicant le préfiltre EST avant d'appliquer le traitement SPT ou LRPT.

Voici un fragment du code, en particulier la méthode `traitementESTSPT` :

```
private Task traitementESTSPT (ArrayList<Task> tachesFaiseables, Instance instance){
      return traitementSPT(traitementEST(tachesFaiseables, instance), instance);
    }
```

3.2 Tableau des résultats obtenus avec analyse des résultats

Maintenant que nous avons vu le code, voici les résultats obtenus :

3.2.1 Le tableau

			basic			greedyspt			greedylpt		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	16	12	9,1	15	16	45,5	0	16	45,5
ft06	6x6	55	0	60	9,1	7	108	96,4	0	112	103,6
ft10	10x10	930	0	1319	41,8	7	2719	192,4	9	2940	216,1
ft20	20x5	1165	0	1672	43,5	0	2717	133,2	0	2586	122,0
AVG	-	-	4,0	-	25,9	7,3	-	116,9	2,3	-	121,8

greedysrpt			greedylrpt			greedyest_lrpt			greedyest_spt		
runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
0	14	27,3	0	11	0,0	1	11	0,0	0	11	0,0
5	140	154,5	2	96	74,5	2	67	21,8	2	88	60,0
3	2670	187,1	2	2089	124,6	8	1112	19,6	1	1074	15,5
17	2242	92,4	0	2223	90,8	11	1474	26,5	3	1267	8,8
6,3	-	115,3	1,0	-	72,5	5,5	-	17,0	1,5	-	21,1

FIGURE 3 – Tableau des résultats obtenu pour la méthode gloutonne

3.2.2 Analyse des résultats

On remarque que mes résultats ne sont pas si différents de ceux fournis dans le fichier `result.txt`. Les différences peuvent s'expliquer par un environnement différent, ou l'utilisation d'égalités de façon différente que la solution fournie

On remarque également que les résultats de `est_lrpt`, sont meilleurs que lrpt, idem pour `est_spt`, explicable via une restriction aux tâches pouvant commencer au plus tôt. Il s'agit donc d'une réduction de l'espace de recherche.

On peut en conclure que les heuristiques gloutonnes sont simples et efficaces (quasi linéaires).

4 Méthode de descente et voisinage

Nous abordons dans ce chapitre la méthode de descente et voisinage, en explictant le principe général de la méthode implémentée via un pseudo-code et enfin le tableau des résultats obtenus avec une analyse des résultats.

4.1 Principe général des méthodes implémentées

Voici un pseudo code de l'algorithme implémenté pour la méthode public `solve`, dans la classe `DescentSolver`. `DescentSolver` essaie de s'appuyer sur l'exploration successive d'un voisinage de solutions.

Ce pseudo code n'est pas exhaustif, et ne tente de refléter que la philosophie générale de l'algorithme.

Algorithm 2: méthode public solve - Méthode de Descente

```
Data: Instance, Deadline
Result: Result
initialisation ← GreedySolver(Priority.EST_LRPT).solve(Instance);
best ← initialisation ;
bool ← True;
while bool and System.currentTimeMillis() < deadline do
    bool ← False;
    order ← ResourceOrder();
    criticalPathBlocks ← blocksOfCriticalPath(order) ;
    foreach block ∈ criticalPathBlocks(best) do
        foreach swap ∈ neighbors(block) do
            temp ← ResourceOrder(best) ;
            applyOn(swap, temp);
            duration ← temp.toSchedule().makespan() ;
            if duration < makespan(best) then
                best ← toSchedule(temp);
                bool ← True;
            else
        end
    end
end
```

Concernant la méthode Descente, comme pour `GreedySolveOr`, j'ai essayé de suivre étape par étape les consignes et explications du sujet de TP.

Dans un premier temps, j'ai complété la méthode `blocksOfCriticalPath` pour extraire la liste des blocs d'un chemin critique puis la méthode `neighbors` pour générer le voisinage d'un bloc, c'est à dire l'ensemble des permutations. J'ai essayé de mettre quelques commentaires, il n'est pas pertinent de reproduire ou expliciter plus en détails ces 2 méthodes.

Concernant le `solve`, nous générons une solution réalisable avec la méthode `est_lrpt` (car elle est utilisé dans le fichier `result.txt`, facilitant le test et la comparaison), puis nous mémorisons la meilleure solution. Dans une boucle while, nous choisissons le meilleur voisin. Si le meilleur voisin est meilleur que notre meilleure solution, alors nous changeons la valeur de la meilleure solution et on reboucle.

On s'arrête s'il n'y a pas d'amélioration -autrement dit on a trouvé un optimum local- ou s'il y a un timeout, c'est à dire si `System.currentTimeMillis()<deadline`.

Le voisinage que nous avons implémenté s'appuie sur la notion de chemin critique et sur la notion de blocs dans le chemin critique. Le calcul d'un chemin critique est fourni dans la classe `Schedule` via la méthode `criticalPath()` qui retourne une liste de tâches. La méthode `makespan()` permet de connaître la durée totale d'une solution. J'ai ainsi permuté les deux tâches en début de bloc et les deux tâches en fin de bloc.

4.2 Tableau des résultats obtenus avec analyse des résultats

Maintenant que nous avons vu le code, voici les résultats obtenus :

4.2.1 Le tableau

			greedyest_lrpt			descent		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	13	11	0,0	15	11	0,0
ft06	6x6	55	3	67	21,8	16	61	10,9
ft10	10x10	930	0	1112	19,6	32	1079	16,0
ft20	20x5	1165	15	1474	26,5	11	1408	20,9
la01	10x5	666	1	699	5,0	2	666	0,0
la02	10x5	655	1	848	29,5	4	813	24,1
la03	10x5	597	0	759	27,1	0	754	26,3
la04	10x5	590	0	782	32,5	0	782	32,5
la05	10x5	593	0	600	1,2	0	593	0,0
la06	15x5	926	0	1141	23,2	0	1135	22,6
la07	15x5	890	0	1122	26,1	3	1122	26,1
la08	15x5	863	0	979	13,4	0	955	10,7
la09	15x5	951	0	1117	17,5	0	1070	12,5
AVG	-	-	2,5	-	18,7	6,4	-	15,6

FIGURE 4 – Tableau des résultats obtenu pour la méthode gloutonne

4.2.2 Analyse des résultats

Ainsi, on obtient 18,7% avec `EST_LRPT` et 15,6% avec la descente appliquée dessus, soit une différence améliorante d'environ 3%. On constate donc que la descente n'améliore pas beaucoup.

L'amélioration est plus significative si on part d'une solution de moins bonne qualité que celle donnée par `EST-LRPT`. Par exemple, si on compare avec `EST_STP`, qui n'a aucune propriété de minimum local est amélioré par `DescentSolver`. Les résultats entre la théorie et la pratique sont cohérents,

En conclusion, la méthode de descente permet une intensification de l'exploration autour d'une solution initiale, s'appuie sur l'exploration successive d'un voisinage de solutions. et aboutit à un optimum local.

5 Méthode Tabou

Ce chapitre explicite la Méthode Tabou, le principe général de la méthode implémentée, le Tableau des résultats obtenus avec une analyse des résultats. Je n'ai malheureusement pas eu le temps de rédiger un pseudo-code pour ce chapitre.

5.1 Principe général des méthodes implémentées

La méthode Tabou est basée sur l'exploration de voisinage et permet de sortir des optima locaux en acceptant des solutions non améliorantes. 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.

Concrètement, j'ai repris `DescentSolver` en modifiant le solver afin de répondre au cahier des charges.

Je reprends donc une solution initiale générée par `GreedySolver`, nommée initialisation. J'ai créé 2 variables supplémentaires, la solution courante et la meilleure solution, initialisation via la solution initiale. Je crée une matrice correspondant aux solutions tabou, et un compteur d'itérations.

On passe ensuite dans une boucle while, explorant les voisinages successifs. Dans cette boucle, on vérifie que le compteur, et la deadline ne sont pas dépassées.

On incrémente le compteur, on choisi le meilleur voisin non tabou que l'on ajoute à la matrice des solutions taboo, et à la variable solution courante. Si, et seulement si ce meilleur voisin est meilleur que la meilleure solution jusqu'ici, alors on la remplace et on reboucle.

On s'arrête uniquement s'il y a TimeOut ou si le compteur est supérieur à MaxIter, tous les deux définis dans le Main.

Le voisinage utilisé est le même que celui implémenté pour la méthode de descente. La seule différence correspond au fait que la solution voisine sélectionnée est la meilleure de tout le voisinage même si elle n'est pas améliorante.

Pour gérer l'ensemble des solutions tabou, il est nécessaire de définir une structure de données permettant de vérifier si une solution a déjà été visitée ou non.

Pour extraire un id entier unique pour une tache (j, i) j'ai utilisé : `j * TASK_PER_JOB + i`

5.2 Tableau des résultats obtenus avec analyse des résultats

Maintenant que nous avons vu le code, voici les résultats obtenus :

5.2.1 Le tableau

			greedyest_lrpt			descent			taboo		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	35	11	0,0	27	11	0,0	4	11	0,0
ft06	6x6	55	0	67	21,8	14	61	10,9	48	61	10,9
ft10	10x10	930	13	1112	19,6	27	1079	16,0	26	1079	16,0
ft20	20x5	1165	12	1474	26,5	1	1408	20,9	16	1408	20,9
la01	10x5	666	0	699	5,0	15	666	0,0	17	666	0,0
la02	10x5	655	0	848	29,5	15	813	24,1	0	813	24,1
la03	10x5	597	0	759	27,1	0	754	26,3	2	754	26,3
la04	10x5	590	0	782	32,5	0	782	32,5	0	782	32,5
la05	10x5	593	0	600	1,2	0	593	0,0	0	593	0,0
la06	15x5	926	0	1141	23,2	0	1135	22,6	45	994	7,3
la07	15x5	890	0	1122	26,1	0	1122	26,1	0	1122	26,1
la08	15x5	863	0	979	13,4	0	955	10,7	0	955	10,7
la09	15x5	951	0	1117	17,5	0	1070	12,5	0	1070	12,5
AVG	-	-	4,6	-	18,7	7,6	-	15,6	12,2	-	14,4

FIGURE 5 – Tableau des résultats obtenu pour la méthode gloutonne

5.2.2 Analyse des résultats

On constate que la différence est de 4%. Même si cela est dépendant de beaucoup de détails d'implémentation et des paramètres choisis (une deadline à 1s et un maxIter à 1000), on s'attendait à descendre normalement sous les 5% d'écart. J'ai effectué de nombreux tests, essayé de changer régulièrement mon code, mais je ne suis pas en mesure d'expliquer pourquoi mon `TabooSolver` n'est pas aussi performant qu'espéré.

On peut néanmoins, en conclusion, confirmer, que même si on n'explore pas toute les solutions et que l'on n'aura aucune garantie d'obtenir la solution optimale, `TabooSolver` reste le plus performant. Il est possible d'améliorer le solver, mais il y a un risque pour que la convergence peut être lente. C'est pourquoi nous devons fixer un nombre maximal d'itérations.

6 Comparaison des différentes méthodes

Comparaison des différentes méthodes

Nous avons dans ce rapport traité de 4 types méthodes :

1. Méthode gloutonne
2. Méthode de descente
3. Méthodes exhaustives
4. Méthode Taboo

Nous avons vu dans le cours qu'il y avait encore d'autres méthodes heuristiques possibles. Je ne pense pas qu'une méthode soit meilleure que les autres. A l'inverse, nous avons vu que certaines méthodes, pour des problèmes données, sont plus efficaces ou plus performantes en fonction des critères données. Par exemple, sur `Taboo`, on remarque que l'écart est considérablement réduit mais qu'à l'inverse le runtime est élevé.

On peut produire le même raisonnement avec la méthode gloutonne, ou LRPT fourni un résultat avec le runtime le plus faible, pour un écart relativement faible par rapport à d'autres règles de priorité.

L'analyse et la comparaison des méthodes se doit donc de reposer sur des critères explicites. C'est à l'utilisateur de définir ces priorités. Souhaite t'il obtenir les résultats le plus rapidement possible, et dans ce cas il devra choisir en priorité `GreelyLRPT`, ou des résultats les meilleurs possibles, et dans ce cas c'est principalement Taboo qu'il faudra privilégier.

En fonction des problèmes, une méthode sera plus efficace que d'autre. Mais les critères de performance repose également sur la manière dont on a codé les algorithmes, les méthodes solve. J'aurais pu travailler pour améliorer la performance non pas des méthodes mais de l'algorithme, réduire sa complexité. C'est un élément important, la manière de coder les égalités etc.

Nous avons également vu, dans ce rapport, les espaces de recherches, et les représentations par numéro de job, par ordre de passage etc dans le cadre des méthodes exhaustives. L'avantage incontestable est qu'on a l'assurance d'obtenir la meilleure solution, mais pour un temps d'exécution particulièrement important.

Conclusion

Ce projet fut très intéressant, car il m'a permis de mettre concrètement en application la théorie vu lors des cours magistraux. A l'issue de ces Travaux Pratiques, j'ai réussi à implémenter l'ensemble des méthodes et solveurs demandés. Je pense néanmoins que `TabooSolver` n'est pas complètement juste car les tests de performance démontrent des résultats inférieur à ce qu'escompté notamment dans le fichier `result.txt`. Par manque de temps, je n'ai pas pu également traiter les questions bonus.

Néanmoins, ce rapport montre bien la multitude de méthodes et d'heuristiques existantes. Nous avons vu que certaines étaient plus pertinentes que d'autres, plus efficaces que d'autres. Nous avons vu leur utilité. Nous avons également vu la pertinence du choix de la représentation d'un problème d'ordonnancement.