

Rapport TPs Metaheuristiques

- Heuristiques gloutonnes
- Méthode de descente
- Méthode Tabou

Imad Boukezzata
MEJRI Hazem

Rapport TPs Metaheuristiques

- Heuristiques gloutonnes
- Méthode de descente
- Méthode Tabou

Imad Boukezzata
MEJRI Hazem

Table des matières

1	Introduction et problème étudié	1
2	Heuristiques gloutonnes et analyse de complexité	2
2.1	Présentation des méthodes implémentées (SPT, LRPT, EST-SPT, EST-LRPT)	2
2.2	Complexité des heuristiques SPT, LRPT et EST-SPT	4
2.3	Analyse des résultats obtenus sur des jeux de données variés et significatifs	5
2.4	Synthèse	5
3	Méthode de descente	6
3.1	Objectifs	6
3.2	Les points importants de la méthode de descente implémentée .	6
3.3	Analyse des résultats	8
3.4	Synthèse	9
4	Méthode Tabou	9
4.1	Objectifs	9
4.2	Les points importants de la méthode de descente implémentée .	9
4.3	Analyse des résultats	10
4.4	Synthèse	11
5	Conclusion	11

1 Introduction et problème étudié

Dans le cadre de notre travail en Systèmes Intelligents, nous avons exploré la résolution de problèmes d'ordonnancement, en mettant l'accent sur le célèbre problème de Job-Shop. Ce défi consiste à minimiser la durée totale nécessaire pour accomplir un ensemble de tâches, réparties sur plusieurs machines. Chaque tâche doit être exécutée sur une machine spécifique, et chaque machine ne peut traiter qu'une tâche à la fois.

Pour résoudre ce problème complexe, nous avons développé et testé différentes méthodes d'optimisation, en utilisant des approches appelées méta-heuristiques. Ces stratégies de résolution, guidées par des heuristiques, ont été implémentées en Java à partir d'une base de code existante, puis évaluées sur divers jeux de données représentant des problèmes de différentes tailles.

En utilisant ces méthodes, notre objectif était d'améliorer progressivement la résolution du problème de Job-Shop en cherchant à minimiser le makespan, c'est-à-dire le temps total nécessaire pour terminer toutes les tâches. Nous décrirons plus en détail les différentes approches explorées dans la suite de ce rapport.

En travaillant sur le problème du Job-Shop, nous avons appliqué ce que nous avons appris dans le cours sur les Systèmes Intelligents. Ça a été une manière concrète de mettre en pratique nos connaissances.

On a utilisé des méthodes un peu avancées, comme des métaheuristiques, pour essayer d'améliorer le temps total de traitement des tâches sur différentes machines. On a testé plusieurs approches pour voir celles qui fonctionnaient le mieux en fonction de la taille des problèmes.

Lien vers notre dépôt git : https://github.com/HAzmej/TP_Methaheuristique

Dans la suite de notre travail, nous nous concentrerons sur une instance particulière, "ft06", pour expliquer la méthode de construction d'un diagramme de Gantt, un outil visuel permettant de représenter la planification des tâches sur les machines. Cette approche nous aidera à déduire le makespan associé à une solution donnée, en visualisant la séquence d'exécution des tâches sur chaque machine.

Cette méthode de construction de diagramme de Gantt se révèle être un outil précieux pour évaluer visuellement la qualité des solutions d'ordonnancement et déterminer rapidement le makespan correspondant.

2 Heuristiques gloutonnes et analyse de complexité

Cette section vise à mettre en œuvre plusieurs approches gloutonnes pour réduire au minimum la durée totale d'un problème de Job-Shop.

L'objectif est de mettre en œuvre un solveur d'ordonnancement de tâches en utilisant différentes politiques de priorité afin d'optimiser l'exécution des tâches sur des machines disponibles. Ces politiques consistent à choisir à chaque étape l'opération ou la tâche qui semble la plus prometteuse en fonction d'une certaine métrique, telles que les durées des opérations restantes, les temps de traitement estimés, ou les marges de temps disponibles pour chaque opération.

2.1 Présentation des méthodes implémentées (SPT, LRPT, EST-SPT, EST-LRPT)

Les détails et explications sont inclus dans le code que nous avons fourni.

```
case SPT:
    Task SPT_task = tabtask.get(0);
    for (Task newtask : tabtask) {
        if (instance.duration(newtask) <= instance.duration(SPT_task)) {
            SPT_task = newtask;
        }
    }
    tabtask.remove(SPT_task);
    if ((SPT_task.task + 1) < nb_tache_per_job) {
        Task task_ajouter = new Task(SPT_task.job, task: SPT_task.task + 1);
        tabtask.add(task_ajouter);
    }
    manualR0.addTaskToMachine(instance.machine(SPT_task), SPT_task);

    SPT_task = null;
    break;
```

FIGURE 1 – SPT

```
case LRPT :
    max=0;
    Task LRPT_task = tabtask.get(0);

    //Durée restantes de tous les jobs
    for (int i = LRPT_task.task ; i <= nb_tache_per_job-1; i++) {
        max += instance.duration(LRPT_task.job, task: i);
    }

    //parcours de toutes les tâches pour trouver la plus longue
    for (Task newtask: tabtask) {
        temps=0;

        //calcul durée restante pour chaque tâche à partir de la tâche actuelle
        for (int i = newtask.task ; i <= nb_tache_per_job-1; i++) {
            temps += instance.duration(newtask.job, i);
        }
        if (temps > max) {
            max=temps;
            LRPT_task=newtask;
        }
    }
    if (LRPT_task!=null) {
        tabtask.remove(LRPT_task);
        //Si task selectionne n'est pas la dernière tâche de son job
        if ((LRPT_task.task + 1) < nb_tache_per_job) {
            //cree prochaine tâche du meme job et l'ajoute a liste des tâches à traiter
            Task task_ajouter = new Task(LRPT_task.job, task: LRPT_task.task + 1);
            tabtask.add(task_ajouter);
        }
        manualR0.addTaskToMachine(instance.machine(LRPT_task), LRPT_task);
    }
    LRPT_task=null;
    break;
```

FIGURE 2 – LRPT

```

case EST_SPT:

    Task EST_SPT_task = tabtask.get(0);
    //check le temps de demarrage de la tache init en prenant le max entre dispo du job et machine

    min = Math.max(dispo_job.get(EST_SPT_task.job),dispo_machine.get(instance.machine(EST_SPT_task)));

    //pour chaque tache on va check le temps par rapport au max entre dispo_job et machine
    for (Task newtask : tabtask){
        temps = Math.max(dispo_job.get(newtask.job),dispo_machine.get(instance.machine(newtask)));

        if (temps<min){
            min=temps;
            EST_SPT_task=newtask;
        }
        // Si on a le meme temps de demarrage
        else if (temps==min){
            //Si la duree de la nouvel tache est plus courte
            if (instance.duration(newtask)<instance.duration(EST_SPT_task)){
                min=temps;
                EST_SPT_task=newtask;
            }
        }
    }
    if (EST_SPT_task!=null) {
        tabtask.remove(EST_SPT_task);

        //Si la tache n'est pas la derniere de son job
        if ((EST_SPT_task.task + 1) < nb_tache_per_job) {
            //cree prochaine tache du meme job et l'ajoute a liste des taches à traiter
            Task task_ajouter = new Task(EST_SPT_task.job, task: EST_SPT_task.task + 1);
            tabtask.add(task_ajouter);
        }
        manualRO.addTaskToMachine(instance.machine(EST_SPT_task), EST_SPT_task);

        dispo_machine.set(instance.machine(EST_SPT_task), min + instance.duration(EST_SPT_task));
        dispo_job.set(EST_SPT_task.job, min + instance.duration(EST_SPT_task));
    }
    break;

```

FIGURE 3 – EST_SPT

```

case EST_LRPT:

    Task EST_LRPT_task = tabtask.get(0);
    max = Math.max(dispo_job.get(EST_LRPT_task.job), dispo_machine.get(instance.machine(EST_LRPT_task)));

    for (Task newtask : tabtask) {
        temps = Math.max(dispo_job.get(newtask.job), dispo_machine.get(instance.machine(newtask)));
        if (temps < max) {
            max = temps;
            EST_LRPT_task = newtask;
        } else if (temps == max) {
            if (instance.duration(newtask) > instance.duration(EST_LRPT_task)) {
                max = temps;
                EST_LRPT_task = newtask;
            }
        }
    }
    if (EST_LRPT_task!=null){
        tabtask.remove(EST_LRPT_task);
        if (EST_LRPT_task.task+1<nb_tache_per_job){
            Task task_ajouter = new Task(EST_LRPT_task.job, task: EST_LRPT_task.task + 1);
            tabtask.add(task_ajouter);
        }
        manualRO.addTaskToMachine(instance.machine(EST_LRPT_task), EST_LRPT_task);

        dispo_machine.set(instance.machine(EST_LRPT_task), max + instance.duration(EST_LRPT_task));
        dispo_job.set(EST_LRPT_task.job, max + instance.duration(EST_LRPT_task));
    }

    break;
}

```

FIGURE 4 – EST_LRPT

2.2 Complexité des heuristiques SPT, LRPT et EST-SPT

- SPT : $O(n)$

Le temps d'exécution est proportionnel au nombre de tâches prêtes à traiter à chaque instant parcours de tabtask pour trouver la tâche la plus courte :

- nombre d'itérations est égal au nombre total de tâches dans le système, qui est n
- Pire des cas : vérifie toutes les tâches restantes, $O(n)$ pour chaque tâche, sauf que les tâches sont ordonnées par rapport au job et donc rajout un $O(\log(n))$ à la complexité

- LRPT : $O(n*m)$ m correspond aux nombre de tâches par job

Temps d'exécution est proportionnel à la somme des durées restantes de toutes les tâches.

- Chaque itération, nous devons trouver la tâche qui a la plus grande durée restante à traiter, ce qui prend donc $O(n)$

```
for (Task newtask : tabtask)
```

```
    temps = 0;
```

```
    //boucle parcourt également tous les éléments de tabtask, donc
```

```
    elle a une complexité  $O(n)$ 
```

```
    for (int i = newtask.task; i < nb_tache_
```

```
        per_job - 1; i++)
```

```
        //boucle parcourt également tous les jobs ( $m$ ) d'une tâche, donc
```

```
        elle a une complexité  $O(m)$ 
```

```
    ...
```

- `tabtask.remove(LRPT_task)` : Complexité $O(n)$ dans le pire cas, car elle supprime un élément spécifique de la liste.

- `manualRO.addTaskToMachine(instance.machine(LRPT_task), LRPT_task)` : Complexité dépendant de l'implémentation de `manualRO` : $O(n)$

- EST_SPT : $O(n)$

- `max = Math.max(dispo_job.get(EST_LPT_task.job), dispo_machine.get(instance.machine(EST_LPT_task)))` : Complexité $O(1)$ (opération simple).

- La boucle for itère sur tabtask, qui contient n tâches..

- À chaque itération :

```
    Temps = Math.max(...); Complexité  $O(1)$  .
```

```
    Les comparaisons et affectations dans les if sont de complexité  $O(1)$  .
```

```
    La boucle entière a donc une complexité de  $O(n)$  .
```

- `tabtask.remove(EST_LPT_task)` : Complexité $O(n)$ pour supprimer un élément spécifique d'une liste.

Donc, la complexité totale de ce code est $O(n) + O(n)$, ce qui simplifie à $O(n)$.

- EST_LRPT : $O(n)$ pareil que EST_SPT

Optimisation : utilisation de méthodes de recherche plus efficaces ou l'optimisation de la suppression d'éléments dans les listes.

2.3 Analyse des résultats obtenus sur des jeux de données variés et significatifs

Pour tester les fonctions SPT, LRPT, EST_SPT et EST_LRPT on a utilisé dans un premier temps l'instance ft pour nous assurer qu'on avait bien les bons résultats

spt						lrpt					
instance	size	best	runtime	makespan	ecart	instance	size	best	runtime	makespan	ecart
ft06	6x6	55	8	124	125,5	ft06	6x6	55	5	74	34,5
ft10	10x10	930	1	3101	233,4	ft10	10x10	930	2	1344	44,5
ft20	20x5	1165	1	3144	169,9	ft20	20x5	1165	2	1955	67,8
AVG	-	-	3,3	-	176,3	AVG	-	-	3,0	-	49,0

est_spt						est_lrpt					
instance	size	best	runtime	makespan	ecart	instance	size	best	runtime	makespan	ecart
ft06	6x6	55	5	88	60,0	ft06	6x6	55	7	60	9,1
ft10	10x10	930	1	1074	15,5	ft10	10x10	930	1	1319	41,8
ft20	20x5	1165	1	1267	8,8	ft20	20x5	1165	2	1672	43,5
AVG	-	-	2,3	-	28,1	AVG	-	-	3,3	-	31,5

Les premiers résultats indiquent qu'EST_LRPT est le solveur le plus rapide parmi tous, avec un écart moyen de 9.1. Nous remarquons également qu'il n'y a pas une grande différence entre EST_LRPT et LRPT. En revanche, la méthode SPT présente le plus grand écart avec une valeur moyenne de 176.3. Nous notons que EST a considérablement amélioré les résultats du SPT.

Ensuite, pour pouvoir déterminer la meilleure solution, nous avons testé avec plusieurs instances plus complexes et nous avons enregistré quelques résultats :

est_spt					
instance	size	best	runtime	makespan	ecart
la01	10x5	666	6	751	12,8
la02	10x5	655	0	821	25,3
la03	10x5	597	0	672	12,6
la04	10x5	590	1	711	20,5
la05	10x5	593	0	610	2,9
la06	15x5	926	0	1200	29,6
la07	15x5	890	0	1034	16,2
la08	15x5	863	0	942	9,2
la09	15x5	951	2	1045	9,9
la10	15x5	958	1	1049	9,5
la11	20x5	1222	1	1473	20,5
la12	20x5	1039	2	1305	25,6
la13	20x5	1150	2	1275	10,9
la14	20x5	1292	2	1427	10,4
la15	20x5	1207	1	1376	14,0
la16	10x10	945	1	1156	22,3
la17	10x10	784	1	924	17,9
la18	10x10	848	1	981	15,7
la19	10x10	842	1	940	11,6
la20	10x10	902	1	1000	10,9
la21	15x10	1046	1	1324	26,6
la22	15x10	927	1	1180	27,3
la23	15x10	1032	1	1162	12,6
la24	15x10	935	1	1203	28,7
la25	15x10	977	1	1449	48,3
la26	20x10	1218	1	1498	23,0
la27	20x10	1235	1	1784	44,5
la28	20x10	1216	1	1610	32,4
la29	20x10	1152	1	1754	52,3
la30	20x10	1355	1	1792	32,3
la31	30x10	1784	1	2001	12,2
la32	30x10	1850	1	2292	23,9
la33	30x10	1719	2	1945	13,1
la34	30x10	1721	1	2070	20,3
la35	30x10	1888	1	2145	13,6
la36	15x15	1268	1	1799	41,9
la37	15x15	1397	1	1669	19,5
la38	15x15	1196	1	1404	17,4
la39	15x15	1233	1	1599	29,7
la40	15x15	1222	1	1476	20,8
AVG	-	-	1,1	-	21,2

Après analyse, nous constatons que sur un ensemble de données étendu :

- L'utilisation de l'algorithme EST permet d'améliorer significativement les performances par rapport à LRPT et SPT.
- Sur ce même ensemble de données, le SPT se distingue par ses résultats les moins satisfaisants et sa durée d'exécution la plus longue.
- Enfin, l'algorithme EST_LRPT se démarque en offrant les résultats les plus optimaux tout en s'exécutant rapidement.

2.4 Synthèse

Nous pouvons conclure à la fin de cette section que l'algorithme EST_LRPT permet d'avoir les meilleurs résultats tout en optimisant le temps d'exécution

3 Méthode de descente

3.1 Objectifs

Dans cette section, notre but est d'implémenter une méthode de descente pour potentiellement améliorer les solutions par rapport aux approches gloutonnes.

À partir de l'ordre des ressources, Il genere des voisins en modifiant l'ordre des tâches. Il tente de trouver une solution optimale en explorant les voisins générés par le voisinage de Nowick en choisissant les solutions avec un makespan plus court jusqu'à ce qu'aucune amélioration ne soit trouvée ou que le délai soit atteint.

3.2 Les points importants de la méthode de descente implémentée

```
List<Block> blocksOfCriticalPath(ResourceOrder order) {
    //recup num de machine
    //commencez par lere task avec sa machine
    //iterer juqu'a trouver une autre task avec la meme machine
    //stocker task1 , task2 et la meme machine

    List<Task> cheminCritique = order.toSchedule().get().criticalPath();
    List<Block> listeDesBlocs = new ArrayList<>();

    Task task = cheminCritique.get(0);

    int machine = order.instance.machine(cheminCritique.get(0));
    int firstTask = Arrays.asList(order.tasksByMachine[machine]).indexOf(task);
    int lastTask = firstTask;

    for (int i = 1; i < cheminCritique.size(); i++) {
        task = cheminCritique.get(i);

        if (machine == order.instance.machine(task)) {
            lastTask++;
        } else {
            if (firstTask != lastTask) {
                listeDesBlocs.add(new Block(machine, firstTask, lastTask));
            }

            machine = order.instance.machine(task);
            firstTask = Arrays.asList(order.tasksByMachine[machine]).indexOf(task);
            lastTask = firstTask;
        }
    }
    return listeDesBlocs;
}
```

FIGURE 5 – Blocks of Critical Path

- D'abord, nous faisons appel à la méthode "**criticalPath()**" pour trouver le chemin critique dans le graphe des dépendances de tâches. Cette méthode utilise l'algorithme de la chaîne critique pour identifier le chemin le plus long dans le graphe, ce qui correspond au temps minimum nécessaire pour achever le projet.
- En parcourant les tâches du chemin critique, chaque tâche est examinée pour trouver les blocs adjacents ayant la même machine.
- Pour chaque tâche, l'algorithme itère sur les tâches suivantes du chemin critique pour détecter les tâches appartenant à la même machine et formant ainsi un bloc.

- Si un bloc est identifié (c'est-à-dire si la première et la dernière tâche du bloc sont différentes), il est ajouté à la liste des blocs.
- Retourne la liste des blocs identifiés.

```
@Override
public Optional<Schedule> solve(Instance instance, long deadline) {

    ResourceOrder R0 = new ResourceOrder(instance);
    List<ResourceOrder> list_R0 ;

    int makespan_init = R0.toSchedule().get().makespan();

    long exec_time=0;
    boolean recherche =true;
    ResourceOrder bestR0=R0;
    ResourceOrder prevR0;
    while((exec_time<deadline)&&(recherche)){
        //meilleur ordre trouve pour le moment
        prevR0=bestR0;
        //à chaque fois on genere les voisins du meilleur ordre obtenu
        list_R0 = neighborhood.generateNeighbors(bestR0);
        for(ResourceOrder R :list_R0){
            if (R.toSchedule().get().isValid() && R.toSchedule().isPresent()){
                int makespan = R.toSchedule().get().makespan();
                if (makespan < makespan_init){
                    makespan_init = makespan;
                    bestR0 = R;
                }
            }
        }
        if (prevR0==bestR0){
            recherche=false;
        }
        exec_time=System.currentTimeMillis();
    }
    return bestR0.toSchedule();
}
```

FIGURE 6 – DescentSolver

- Cette méthode DescentSolver tente de trouver une solution optimale en utilisant une recherche de descente.
- Elle prend en paramètres une instance du problème et une limite de temps d'exécution ("deadline").
- Elle commence par initialiser une "ResourceOrder" initiale "R0" pour représenter la première solution.
- Elle obtient le "makespan" initial en calculant le temps nécessaire pour terminer toutes les tâches dans l'ordonnancement initial "R0".
- Ensuite, elle initialise des variables pour suivre le temps d'exécution et l'état de la recherche.
- La boucle principale de la recherche continue tant que le temps d'exécution est inférieur à la limite de délai et qu'une amélioration est possible.
- À chaque itération, elle génère les voisins de la meilleure solution actuelle "bestR0" en utilisant le voisinage défini par "neighborhood".
- Pour chaque voisin généré, elle vérifie s'il est valide et présent dans le "Schedule" et s'il améliore le "makespan".
- Si une meilleure solution est trouvée, elle met à jour la meilleure solution "bestR0" et le "makespan" initial.
- Finalement, elle retourne l'ordonnancement correspondant à la meilleure solution trouvée sous forme de "Schedule".
- **CONDITION D'ARRÊT** : La recherche s'arrête lorsque le délai est atteint ou lorsqu'aucune amélioration n'est trouvée après avoir exploré tous les voisins.

3.3 Analyse des résultats

descente_spt						descente_lrpt					
instance	size	best	runtime	makespan	ecart	instance	size	best	runtime	makespan	ecart
ft06	6x6	55	26	89	61,8	ft06	6x6	55	16	58	5,5
ft10	10x10	930	101	1721	85,1	ft10	10x10	930	30	1111	19,5
ft20	20x5	1165	70	1535	31,8	ft20	20x5	1165	20	1625	39,5
AVG	-	-	65,7	-	59,5	AVG	-	-	22,0	-	21,5
descente_est_spt						descente_est_lrpt					
instance	size	best	runtime	makespan	ecart	instance	size	best	runtime	makespan	ecart
ft06	6x6	55	21	72	30,9	ft06	6x6	55	13	60	9,1
ft10	10x10	930	33	1017	9,4	ft10	10x10	930	33	1178	26,7
ft20	20x5	1165	1	1267	8,8	ft20	20x5	1165	2	1662	42,7
AVG	-	-	18,3	-	16,3	AVG	-	-	16,0	-	26,1

Les méthodes basées sur SPT (descente est_spt et descente spt) montrent que l'heuristique SPT est plus performante lorsqu'elle est combinée avec une stratégie de descente guidée par le temps de fin estimé (EST)

D'après les résultats, descente est_spt est la meilleure règle de descente en termes de compromis entre runtime et écart par rapport au meilleur makespan possible.

- **Runtimes** : Plus rapides avec une moyenne de 11,7 secondes.
- **Écarts** : Moins importants avec une moyenne de 15,5, ce qui signifie que les solutions trouvées sont plus proches des meilleures solutions connues.

descente_est_lrpt						descente_est_spt					
instance	size	best	runtime	makespan	ecart	instance	size	best	runtime	makespan	ecart
la01	10x5	666	12	858	28,8	la01	10x5	666	32	686	3,0
la02	10x5	655	12	813	24,1	la02	10x5	655	12	685	4,6
la03	10x5	597	5	747	25,1	la03	10x5	597	3	666	11,6
la04	10x5	590	6	815	38,1	la04	10x5	590	4	702	19,0
la05	10x5	593	7	604	1,9	la05	10x5	593	2	610	2,9
la06	15x5	926	8	944	1,9	la06	15x5	926	9	963	4,0
la07	15x5	890	10	1014	13,9	la07	15x5	890	2	1034	16,2
la08	15x5	863	11	932	8,0	la08	15x5	863	3	933	8,1
la09	15x5	951	2	992	4,3	la09	15x5	951	5	975	2,5
la10	15x5	958	1	958	0,0	la10	15x5	958	2	1049	9,5
la11	20x5	1222	3	1297	6,1	la11	20x5	1222	8	1390	13,7
la12	20x5	1039	6	1096	5,5	la12	20x5	1039	21	1044	0,5
la13	20x5	1150	2	1150	0,0	la13	20x5	1150	9	1255	9,1
la14	20x5	1292	1	1292	0,0	la14	20x5	1292	2	1427	10,4
la15	20x5	1207	5	1417	17,4	la15	20x5	1207	10	1345	11,4
la16	10x10	945	11	1114	17,9	la16	10x10	945	8	1129	19,5
la17	10x10	784	13	870	11,0	la17	10x10	784	4	905	15,4
la18	10x10	848	11	953	12,4	la18	10x10	848	2	981	15,7
la19	10x10	842	13	986	17,1	la19	10x10	842	4	909	8,0
la20	10x10	902	26	1095	21,4	la20	10x10	902	11	959	6,3
la21	15x10	1046	32	1225	17,1	la21	15x10	1046	12	1257	20,2
la22	15x10	927	6	1275	37,5	la22	15x10	927	6	1169	26,1
la23	15x10	1032	17	1179	14,2	la23	15x10	1032	8	1141	10,6
la24	15x10	935	18	1128	20,6	la24	15x10	935	15	1149	22,9
la25	15x10	977	17	1187	21,5	la25	15x10	977	23	1293	32,3
la26	20x10	1218	24	1488	22,2	la26	20x10	1218	15	1429	17,3
la27	20x10	1235	14	1645	33,2	la27	20x10	1235	18	1705	38,1
la28	20x10	1216	12	1458	19,9	la28	20x10	1216	4	1541	26,7
la29	20x10	1152	19	1442	25,2	la29	20x10	1152	19	1647	43,0
la30	20x10	1355	8	1677	23,8	la30	20x10	1355	6	1666	23,0
la31	30x10	1784	26	2036	14,1	la31	30x10	1784	33	1827	2,4
la32	30x10	1850	72	2047	10,6	la32	30x10	1850	28	2179	20,8
la33	30x10	1719	15	2001	16,4	la33	30x10	1719	10	1917	11,5
la34	30x10	1721	49	1961	13,9	la34	30x10	1721	20	2009	16,7
la35	30x10	1888	14	2339	23,9	la35	30x10	1888	7	2108	11,7
la36	15x15	1268	29	1531	20,7	la36	15x15	1268	31	1694	33,6
la37	15x15	1397	36	1701	27,5	la37	15x15	1397	17	1649	10,0
la38	15x15	1196	73	1439	20,3	la38	15x15	1196	24	1372	14,7
la39	15x15	1233	35	1635	32,6	la39	15x15	1233	5	1599	29,7
la40	15x15	1222	41	1544	26,4	la40	15x15	1222	16	1377	12,7
AVG	-	-	18,1	-	17,4	AVG	-	-	11,7	-	15,5

Les résultats exposés dans le tableau étendu fournissent une perspective plus robuste, étant basés sur un jeu de données plus vaste. On constate que les performances des algorithmes "descente est_lrpt" et "descente est_spt" sont étroitement comparables pour la majorité des instances. Néanmoins, une observation notable émerge pour les instances les plus complexes. Dans ces cas, "descente est_spt" démontre une légère amélioration en termes de temps d'exécution et de makespan.

Il convient de souligner que ces résultats ne permettent pas de généraliser ou de conclure de manière définitive sur la performance des algorithmes étudiés. Il est tout à fait possible que certains algorithmes se révèlent nettement plus efficaces que d'autres dans des cas spécifiques. De plus, la performance de ces algorithmes peut être influencée par les paramètres spécifiques utilisés pour

chaque instance du problème. Par conséquent, il est crucial de poursuivre les tests et les analyses sur un large éventail d'instances et de paramètres afin de déterminer de manière plus précise les performances réelles de ces algorithmes.

Cette analyse suggère que pour les problèmes de jobshop, utiliser une stratégie de descente basée sur l'heuristique SPT avec une évaluation basée sur le temps de fin estimé est optimal pour obtenir des résultats rapides et de haute qualité.

3.4 Synthèse

Les résultats obtenus précédemment avec la méthode de descente sont nettement supérieurs à ceux obtenus par l'heuristique gloutonne sans descente. En effet, pour toutes les instances, l'heuristique gloutonne sans descente a produit des makespans beaucoup plus élevés, accompagnés d'écarts moyens également plus importants.

4 Méthode Tabou

4.1 Objectifs

L'objectif de ce projet est de développer une métaheuristique de type recherche Tabou pour sortir des optima locaux. Cette méthode s'appuie sur l'exploration de voisinage établi dans l'algorithme de descente et permet d'éviter de boucler sur des solutions déjà visitées en utilisant une liste tabou pour interdire.

4.2 Les points importants de la méthode de descente implémentée

Dans la partie du code correspondant à l'étape 8, l'algorithme explore l'espace des solutions en générant des voisins de la solution actuelle et en sélectionnant le meilleur voisin non tabou comme prochaine solution. Voici une explication détaillée de cette partie :

1. Exploration des voisins : - La liste 'neighbors' est initialisée pour stocker les voisins de la solution actuelle 'R0'. Ces voisins sont générés à l'aide de l'objet 'Nowicki'.

- Chaque voisin potentiel est évalué pour déterminer s'il améliore la solution actuelle et s'il n'est pas tabou.

2. Sélection du meilleur voisin :

- Une boucle parcourt tous les voisins générés.

- Pour chaque voisin potentiel, sa validité est vérifiée d'abord. S'il n'est pas valide (c'est-à-dire s'il n'a pas de planification associée), il est ignoré.

- Ensuite, pour chaque voisin valide, sa faisabilité est évaluée en comparant son makespan (temps total d'exécution) avec celui du meilleur voisin trouvé jusqu'à présent. Si le voisin a un makespan inférieur au meilleur makespan trouvé, il est potentiellement une meilleure solution.

- Si le voisin est valide et n'est pas tabou, et s'il améliore le makespan actuel, il est sélectionné comme meilleur voisin et sa valeur de makespan est mise à jour pour comparaison ultérieure.

3. Mise à jour de la solution courante :

- Si un meilleur voisin a été trouvé, la solution actuelle 'R0' est mise à jour avec ce meilleur voisin. - Le makespan correspondant est également mis à jour.

4. Gestion de la liste tabou :

- Une fois le meilleur voisin sélectionné, il est ajouté à la liste tabou 'tabooOrders'.
- L'itération actuelle est également ajoutée à la liste 'tabooIterations', indiquant à quelle itération cette solution est devenue tabou.

5. Répétition jusqu'à la fin de l'algorithme :

- Ce processus de sélection de voisins et de mise à jour de la solution actuelle se répète jusqu'à ce que le nombre maximal d'itérations soit atteint ou que le délai imparti soit dépassé.

Cette approche permet d'explorer efficacement l'espace des solutions en évitant les solutions déjà explorées et en se concentrant sur celles qui améliorent la solution actuelle.

```
@Override
public Optional<Schedule> solve(Instance instance, long deadline) {
    Optional<Schedule> sched = this.BaseSolver.solve(instance, deadline);
    if (!sched.isPresent()) {
        return Optional.empty();
    }

    ResourceOrder bestR0 = new ResourceOrder(sched.get());
    ResourceOrder R0 = bestR0;
    Nowicki now = new Nowicki();

    List<ResourceOrder> tabooOrders = new ArrayList<>();
    List<Integer> tabooIterations = new ArrayList<>();

    int bestMakespan = bestR0.toSchedule().get().makespan();

    for (int iteration = 0; iteration < maxIterations && System.currentTimeMillis() < deadline; iteration++) {
        List<ResourceOrder> neighbors = now.generateNeighbors(R0);
        ResourceOrder bestNeighbor = null;
        int bestNeighborMakespan = Integer.MAX_VALUE;

        for (ResourceOrder neighbor : neighbors) {
            if (!neighbor.toSchedule().isPresent()) {
                continue;
            }
            int neighborMakespan = neighbor.toSchedule().get().makespan();

            boolean isTaboo = false;
            for (int i = 0; i < tabooOrders.size(); i++) {
                if (tabooOrders.get(i).equals(neighbor) && (iteration - tabooIterations.get(i) <= tabooDuration)) {
                    isTaboo = true;
                    break;
                }
            }

            if (!isTaboo && (bestNeighbor == null || neighborMakespan < bestNeighborMakespan)) {
                bestNeighbor = neighbor;
                bestNeighborMakespan = neighborMakespan;
            }
        }

        if (bestNeighbor != null) {
            R0 = bestNeighbor;
            int currentMakespan = bestNeighborMakespan;

            if (currentMakespan < bestMakespan) {
                bestR0 = R0;
                bestMakespan = currentMakespan;
            }

            tabooOrders.add(R0);
            tabooIterations.add(iteration);
        }
    }

    return bestR0.toSchedule();
}
```

4.3 Analyse des résultats

Pour analyser les résultats obtenus, nous avons procédé à une série de tests en variant les valeurs de 'maxIter' ainsi que la durée du tabou. Ces tests ont été effectués sur l'ensemble des instances "la34", en utilisant l'algorithme SPT comme base.

la34	Max_iter=50	Max_iter=100	Max_iter=200
Durée taboo = 1	Make span = 1951 AVG ecart = 13.4	Make span = 1951 AVG ecart = 13.4	Makespan = 1951 AVG ecart = 13.4
Durée taboo = 20	Make span = 1912 AVG ecart = 11.1	Make span = 1859 AVG ecart = 8.0	Make span = 1816 AVG ecart = 5.5
Durée taboo = 35	Make span = 1912 AVG ecart = 11.1	Make span = 1859 AVG ecart = 8.0	Make span = 1816 AVG ecart = 5.5

4.4 Synthèse

Les résultats montrent que la méthode Tabou obtient les meilleures performances en termes de Makespan et d'écart moyen. Cependant, elle nécessite significativement plus de temps pour s'exécuter, environ 22 fois plus longtemps que la méthode de descente. Cela s'explique par le fait qu'elle explore davantage d'options, ce qui demande plus de temps pour parcourir ses choix.

5 Conclusion

Pour conclure, ce rapport sur les métaheuristiques appliquées au problème de Job Shop nous a permis d'explorer et d'analyser diverses approches heuristiques et métaheuristiques. Nous avons commencé par présenter des heuristiques gloutonnes comme SPT, LRPT, EST-SPT et EST-LRPT, en détaillant leur complexité. Les résultats expérimentaux ont montré que l'efficacité de ces heuristiques varie selon les jeux de données, ce qui souligne l'importance de choisir la méthode en fonction des caractéristiques spécifiques du problème.

Nous avons également examiné la méthode de descente mais aussi la méthode Tabou comme une alternative robuste pour échapper aux minima locaux. Les résultats obtenus en utilisant cette méthode, à partir des meilleures heuristiques gloutonnes, ont confirmé son potentiel pour améliorer la qualité des solutions dans des espaces de recherche complexes.

