



**Article 2: Efficient heuristics and metaheuristics for the unrelated  
parallel machine scheduling  
problem with release dates and setup times**



Sheril Kuete Hanfo - Chloé Larroze  
**EI23**

1er Oct. - 30 Oct. 2025

# Table des matière

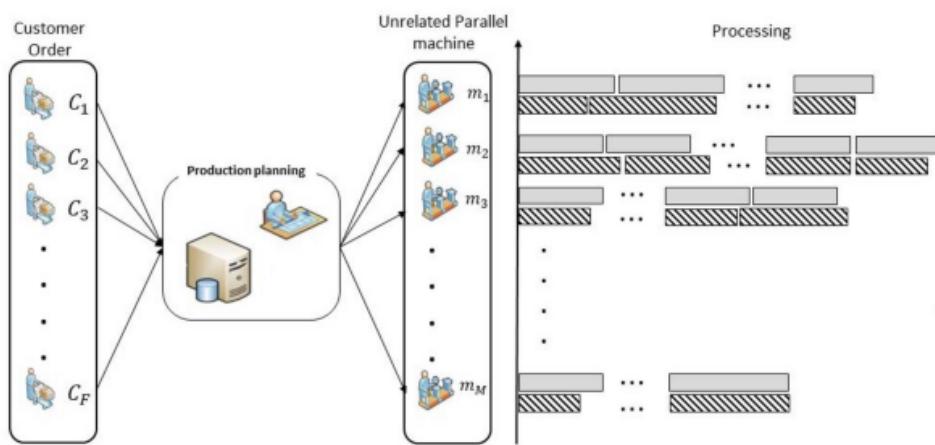
<b>Introduction</b> .....	4
<b>I. Quickstart</b> .....	5
<b>II. Implémentation des Méthodes de Résolution en Java</b> .....	7
A. Compréhension et Synthèse de l'Article.....	7
1. Présentation du problème étudié.....	7
2. Modèle mathématique.....	7
3. Méthodes de résolution proposées.....	8
a) Heuristiques Constructives.....	8
b) LAHC.....	9
c) Résultats comparatifs article.....	10
B. Conception UML.....	10
1. Identification des entités principales.....	10
a) Classes principales.....	10
b) classes “algorithme”.....	11
c) classes “helpers”.....	11
2. Relations entre classes.....	11
a ) Héritage.....	12
b) Compositions & Dépendances.....	12
c) Agrégation.....	12
3. Diagramme UML.....	13
C. Implémentation en Java.....	13
1. Structure du projet Java.....	13
2. Développement de l'algorithme.....	15
a) Structure des Données “fondamentales” - ①.....	15
b) Neighbour - ②.....	18
c) LocalSearch : amélioration par recherche locale - ③.....	19
d) Le Mécanisme d'acceptation de LAHC.....	21
e) Mise à jour de la meilleure solution.....	22
f) Gestion de la liste H.....	23
g) Condition d'arrêt.....	24
3. Tests et validation.....	24
a) test/ : Tests unitaires.....	24
b) Lecture / écriture des instances d'un fichier.....	25
c) Main.java.....	25
d) Problèmes rencontrés.....	26
i) LocalSearch.....	26
ii) Liste historique.....	27
iii) Temps de setups.....	27
e) Résultat.....	27
D. Validation et Analyse des Résultats.....	28
1. Jeux de données utilisés (benchmarks).....	28
a) Premier benchmark: 100 “petites instances”.....	29
b) Second benchmark: 100 instances moyennes.....	31
2. Analyse critique.....	32
a) Efficacité et performance.....	32

b) Limites et pistes d'amélioration.....	32
<b>III. Développement d'une Interface Graphique.....</b>	<b>33</b>
A. Conception de l'Interface.....	33
a) Main.....	33
b) Gantt.....	34
B. Résultat.....	34
Conclusion.....	35
<b>Bibliographie.....</b>	<b>36</b>
<b>Annexes.....</b>	<b>37</b>
A1 - Exemple : un autre problème d'ordonnancement.....	37
A2 - Synthèse de l'article.....	38
A3 - Pourquoi pas MVC ?.....	39
A4 - La taille de l'instance joue-t-elle sur le temps d'exécution ?.....	39



## Introduction

Le problème étudié dans ce rapport, noté  $R |r_j, s_{ijk}| C_{max}$  selon la notation de Graham, combine plusieurs contraintes qui complexifient considérablement sa résolution et qui nécessiteront le développement d'approches heuristiques et/ou métahéuristiques pour obtenir des solutions proches en un temps raisonnable. L'article de référence de *Athmani et al. (2022)* propose à cet effet une étude comparative de plusieurs méthodes de résolution dont l'objectif est de minimiser le makespan, c'est-à-dire le temps d'achèvement du dernier job traité. Ce type de problème trouve ses applications dans de nombreux secteurs tels que la fabrication de semi-conducteurs, l'industrie automobile, l'impression ou encore la construction navale.



Dans le cadre de ce travail, nous nous concentrerons sur l'implémentation en Java de la métahéuristique Late Acceptance Hill Climbing (aussi abrégée LAHC dans la suite de ce rapport). Ce rapport s'organisera comme suit : nous présenterons d'abord une synthèse de l'article de référence, incluant la formalisation mathématique du problème et l'analyse des méthodes proposées. Nous détaillerons ensuite l'implémentation Java de l'algorithme LAHC, en expliquant les choix de conception et les structures de données utilisées. Enfin, nous présenterons les résultats expérimentaux obtenus sur un ensemble d'instances de test et comparerons nos performances avec celles rapportées dans l'article original.

L'ensemble des algorithmes présentés dans ce projet ont été développés en Java 11 et exécutés avec OpenJDK 25 (Zulu build incluant JavaFX) sur un MacBook Pro équipé d'un processeur Intel Core i9 6 cœurs @ 2,9 GHz et 32 Go de RAM DDR4 @ 2400 MHz.

## I. Quickstart

Il s'agit d'une version alégée du quickstart complet trouvable dans le [README.md](#).

### Prérequis

- Java 11 ou supérieur avec JavaFX (le projet utilise Zulu JDK 25 qui inclut JavaFX)
- Terminal/ligne de commande

### Lancement de l'Application

#### Interface Graphique (recommandé)

```
cd LAHC-ParallelMachineScheduling  
chmod +x build.sh  
./build.sh
```

L'interface graphique s'ouvre automatiquement.

#### Ligne de Commande (pour main)

```
./main.sh
```

Compile et exécute le programme en mode console avec une instance aléatoire.

#### Ligne de commande (pour tests unitaires)

```
./tests.sh
```

### Utilisation de l'Interface

#### \* Choisir le Type d'Instance:

- Instance aléatoire : Génère automatiquement une instance avec entre 5-10 jobs et entre 2-4 machines
- Depuis un fichier : Charge l'instance depuis resources/Instance.txt (format spécifique)

\* Lancer la résolution en cliquant sur "Lancer la résolution". L'algorithme s'exécute (généralement < 1 seconde pour petites instances)

\* Les résultats apparaissent dans la zone de texte :

- Nombre de jobs et machines
- Ordonnancement par machine
- Makespan final (temps total de production)

\* Après résolution, le bouton "Afficher le Gantt" devient actif, cliquer dessus.

### Modifier les Paramètres

#### Instance Aléatoire

Dans MainFX.java , lignes 73-74 :

```
java  
int randomJobNumber = 5 + (int)(Math.random() * 6); // 5-10 jobs  
int randomMachineNumber = 2 + (int)(Math.random() * 3); // 2-4 machines
```

Ajustez les valeurs pour tester différentes tailles d'instances.

### Algorithme LAHC

Dans [algo/metaheuristic/LAHCMetaheuristic.java](#):

```
java  
private int historyLength; // Taille de la liste L (défaut: jobs × machines)  
private int maxIterations; // Nombre max d'itérations sans amélioration
```

### Format de Fichier d'Instance

Le fichier [resources/Instance.txt](#) suit ce format :

```
<nombre_jobs> <nombre_machines>  
<p_00> <p_01> ... <p_0m> # Temps de traitement job 0  
<p_10> <p_11> ... <p_1m> # Temps de traitement job 1  
...  
<r_0> <r_1> ... <r_n> # Release dates  
<s_000> <s_001> ... # Setup times (matrice 3D aplatie)
```



## II. Implémentation des Méthodes de Résolution en Java

### A. Compréhension et Synthèse de l'Article

#### 1. Présentation du problème étudié

Le domaine d'application de ce travail concerne ici l'ordonnancement de la production industrielle, et plus précisément le problème  $R | r_j, s_{ijk} | C_{max}$ , aussi reconnu sous le nom de *Unrelated Parallel Machine Scheduling*, avec dates de disponibilité et temps de setup dépendants. Ce type de problème apparaît, selon l'article, dans de nombreux contextes industriels tels que l'impression, la fabrication de semi-conducteurs, l'industrie automobile ou encore la construction navale, où plusieurs machines différentes doivent traiter un ensemble de tâches selon des contraintes complexes.

Par ailleurs, le problème est dit NP-difficile<sup>1</sup>, même dans sa forme la plus simple, ce qui signifie qu'il n'existe pas d'algorithme exact efficace pour le résoudre à grande échelle. Les enjeux sont donc multiples : il s'agit principalement de minimiser le makespan, c'est-à-dire le temps d'achèvement du dernier job, tout en respectant les dates de disponibilité des tâches et en prenant en compte les temps de setup, qui dépendent à la fois de la machine choisie et de la séquence d'exécution.

De plus, ces recherches souhaiteraient également aboutir à une solution permettant de réduire l'écart entre les approches développées et leur application concrète dans l'industrie, afin de proposer des méthodes d'ordonnancement performantes, réalistes mais surtout adaptées aux contraintes opérationnelles réelles.

#### 2. Modèle mathématique

Le problème étudié consiste à ordonner  $n$  jobs sur  $m$  machines parallèles non apparentées. Chaque job  $j$  est caractérisé par une date de disponibilité  $r_j$ , qui correspond au moment à partir duquel il peut commencer à être traité, ainsi que par un temps de traitement  $p_{jk}$  qui dépend de la machine  $k$  sur laquelle il est exécuté. De plus, des temps de setup  $s_{ijk}$  sont associés à chaque paire de jobs  $(i,j)$  sur chaque machine  $k$  ; ils représentent le temps nécessaire pour préparer la machine  $k$  au traitement du job  $j$  après le job  $i$ .

---

<sup>1</sup> En théorie de la complexité, un problème est dit NP-difficile s'il est au moins aussi difficile que les problèmes de la classe NP (Non-déterministe Polynomial time). Cela signifie qu'il n'existe actuellement aucun algorithme exact connu capable de le résoudre en temps polynomial, et qu'une solution efficace pour ce type de problème permettrait de résoudre tous les problèmes NP en temps polynomial. Formellement, si un problème NP-difficile admettait un algorithme en temps polynomial, alors P = NP. Dans la pratique, la résolution exacte devient rapidement inenvisageable dès que la taille du problème augmente.

Le problème est soumis à plusieurs contraintes :

- chaque machine ne peut traiter qu'un seul job à la fois, sans préemption (càd priorité)
- le setup d'un job ne peut commencer avant sa date de disponibilité  $r_j$ . Les temps de setup sont asymétriques, c'est-à-dire que en général  $s_{ijk} \neq s_{jik}$ .
- Enfin, les temps de traitement varient selon les machines, ce qui reflète la nature non apparentée des machines.

L'objectif est donc ici de minimiser le makespan  $C_{max}$ , que l'on peut aussi écrire :

$$C_{max} = \max_k(C_k)$$

défini comme le temps d'achèvement de la machine la plus tardive, où les  $C_k$  désignent le temps d'achèvement de la machine k (cf [\[Annexe A1\]](#)).

### 3. Méthodes de résolution proposées

L'article étudié propose une combinaison de trois heuristiques constructives et de trois métahéuristiques pour résoudre le problème. Dans le cadre de ce travail, nous avons choisi d'implémenter la métahéuristiche LAHC.

#### a) Heuristiques Constructives

Avant d'aborder plus en détails la métahéuristiche choisie, il est nécessaire d'aborder brièvement les trois heuristiques présentées pour générer les solutions initiales. Nous avons donc;

- BIBA (Best-Insertion-Based Approach) : approche déterministe qui insère successivement les jobs dans la position en minimisant le makespan. Elle présente les meilleures performances globales sur l'ensemble des benchmarks, c'est donc vers elle que se portera notre choix pour le calcul des valeurs initiales.
- GRASP (Greedy Randomized Adaptive Search) : méthode aléatoire-gloutone qui construit une solution en sélectionnant de manière probabiliste parmi les meilleures insertions possibles à chaque étape.
- Meta-RaPS : proche de GRASP, mais avec un paramètre contrôlant la probabilité de choisir la meilleure insertion ou une insertion aléatoire, afin d'équilibrer exploitation et exploration.

Ces heuristiques servent à fournir un point de départ pour les métahéuristiques, notamment LAHC et les variantes de recuit simulé.

## b) LAHC

Abordons à présent un peu plus en détail le principe de notre métaheuristique. Il s'agit tout d'abord d'une extension du Hill Climbing<sup>2</sup> classique et repose sur une "liste historique" H de taille LH dans laquelle sont mémorisées les valeurs de la fonction objectif des solutions rencontrées au cours des LH dernières itérations. À chaque itération, une solution voisine est générée aléatoirement à l'aide de deux opérateurs, choisis avec une probabilité égale (soit 50%) :

- Swap interne aléatoire : échange de deux jobs au sein d'une même machine,
- Swap externe aléatoire : échange de deux jobs appartenant à deux machines différentes.

Ainsi, la nouvelle solution est acceptée si :

- elle améliore la solution courante, ou
- elle est meilleure que la solution obtenue il y a LH itérations (ce qui permet d'échapper aux minima locaux)

La méthode intègre ensuite plusieurs opérateurs de recherche locale pour améliorer la solution :

- Bottleneck internal swap
- Bottleneck external insertion
- Bottleneck external swap
- Balancing
- Inter machine insertion

Pour tenter d'en évaluer les capacités et le comparer aux autres méthodes, on peut par ailleurs dresser le tableau avantage/inconvénient suivant:

Avantages	Limites
Simple (un seul paramètre : LH = 30)	ne considère que 2 types de mouvements random pour générer les voisins
Meilleure performance sur les petites instances	Moins performant que SA sur les instances moyennes/grandes
temps de convergence rapide (<5% du temps total)	performance qui se dégrade avec l'augmentation du nombre de jobs

<sup>2</sup> La méthode *hill-climbing* (ou méthode d'escalade) est une procédure d'optimisation itérative qui explore les configurations voisines d'une solution courante et se déplace vers la meilleure d'entre elles jusqu'à atteindre un optimum local, sans garantie d'atteindre l'optimum global.

Évite les minima locaux grâce à l'historique

### c) Résultats comparatifs article

Par ailleurs, on peut citer les résultats mis en évidence dans l'article, où les méthodes ont été évaluées sur un benchmark de 1620 instances réparties en trois ensembles :

- Small (640 instances) : 10 à 40 jobs, 2 à 8 machines
- Medium (180 instances) : 40 à 120 jobs, 2 à 8 machines
- Large (800 instances) : 200 à 1000 jobs, 2 à 8 machines

Les performances de LAHC sont résumées ainsi :

- Petites instances : RPD = 0.05 %, meilleure météahuristique.
- Moyennes instances : RPD = 0.58 %, moins performante que les variantes simulé.
- Grandes instances : RPD = 0.10 %, moins performante que SA mais reste tout de même efficace, raison pour laquelle nous l'avons choisi.

En somme, bien que la LAHC soit moins performant sur les grandes instances, aucune météahuristique ne semble dominer sur tous les types d'instances (cf [\[Annexe A2\]](#)).



## B. Conception UML

Dans cette section, nous présentons la structure conceptuelle de notre solution sous la forme de classes UML. L'objectif ici n'est pas encore d'analyser le fonctionnement détaillé de l'algorithme, mais plutôt de traduire le pseudo-code en un ensemble cohérent d'entités. Cette étape consistera ainsi à identifier les éléments significatifs du problème et à les modéliser sous forme de classes, afin de poser les bases de l'implémentation qui suivra.

### 1. Identification des entités principales

#### a) Classes principales

La première étape est d'analyser les données manipulées par l'algorithme. Pour rappel, le problème mettait en jeu l'ensemble des données suivantes:

- des jobs à ordonner,
- des machines qui ont des vitesses différentes,
- des temps de setup et dates de disponibilité,
- une solution candidate (qui affecte des jobs à des machines dans un certain ordre),
- et un ordonnancement avec des temps de début et de fin.

Ces éléments vont donc naturellement devenir nos classes :

- **Job** : représente un travail à ordonner

- **Machine** : représente une machine de production
- **Instance** : contient toutes les données d'un problème (notamment les temps de setup et dates de disponibilité)
- **Solution** : représente une solution (affectation jobs → machines)
- **Schedule** : représente l'ordonnancement détaillé sur une machine

### b) classes “algorithme”

Ensuite, on regarde les méthodes utilisées pour résoudre le problème. L'article présentait notamment:

- des heuristiques pour générer les solutions initiales (BIBA, GRASP, Meta-RaPS),
- une métaheuristique LAHC pour améliorer cette solution,
- des opérateurs de voisinage pour “explorer” l'espace des solutions.

Ces éléments se traduisent donc en classes responsables des différentes stratégies que nous allons explorer. Comme nous avons pu le voir en cours d' **“Application de Bases de données”**, le Java fonctionne en interface/implémentation, c'est-à-dire une description avec les méthodes et prototypes d'une part puis l'implémentation de ces derniers dans un second temps. On aura donc deux interfaces puis leurs implémentations:

- **Heuristic** (interface) : Interface pour les heuristiques constructives
- **BIBAHeuristic** : Implémentation de l'heuristique BIBA
- **LocalSearch** : Opérateurs de recherche locale (pseudo-code)
- **LAHCMetaheuristic** : Implémentation de Late Acceptance Hill Climbing
- **NeighborhoodOperator** (interface) : Interface pour les opérateurs de voisinage
- **RandomExternalSwap & RandomInternalSwap** : Implémentations pour les opérateurs de voisinages (externe et interne séparément)

Ces classes permettent de séparer la logique du problème lui-même (càd nos classes principales) de la logique algorithmique (càd de la partie résolution), ce qui facilitera les tests ou ajouts éventuels futurs.

### c) classes “helpers”

Enfin, nous avons dû, à posteriori de la modélisation UML, développer un ensemble de classes faisant fonctionner le programme “autour de l'algorithme”. Elles existent bien, mais elles ne découlent pas directement de l'analyse du problème d'ordonnancement, et contiennent des ajouts techniques nécessaires à l'implémentation logicielle (chargement, sauvegarde, évaluation). Elles ne sont donc pas centrales dans la modélisation conceptuelle de notre problème, raison pour laquelle nous ne les représenterons pas forcément dans notre modélisation.

## 2. Relations entre classes

L'architecture orientée objet retenue repose sur différents types de relations entre les classes : héritage, composition forte, agrégation et dépendances. Ces relations structurent

l'interaction entre les entités représentant le problème d'ordonnancement et les entités algorithmiques chargées de le résoudre.

### a) Héritage

L'héritage est utilisé principalement pour définir des interfaces communes à plusieurs stratégies. Ainsi, la classe abstraite `Heuristic` définit le contrat que doivent respecter toutes les heuristiques constructives. Elle est implémentée par `BIBAHeuristic`, qui fournit une méthode gloutonne de construction de solution initiale.

De la même manière, `NeighborhoodOperator` définit une interface commune pour les opérateurs de voisinage utilisés par la métahéuristique LAHC. Puis, nous pouvons implémenter nos swaps, avec deux implementations de prévues : `RandomInternalSwap`, qui effectue un échange de deux jobs sur une même machine, et `RandomExternalSwap`, qui échange deux jobs entre deux machines différentes.

### b) Compositions & Dépendances

Illustrons à présent les compositions, c'est-à-dire, si l'objet conteneur est détruit, les objets contenus le seront également. C'est le cas de la classe `Instance`, qui compose un tableau de `Job` et de `Machine`. Les jobs et les machines sont définis dans le cadre d'une instance donnée et n'ont pas d'existence indépendante en dehors de celle-ci. Cette relation reflète directement la structure d'un problème d'ordonnancement, dans lequel les jobs et les machines sont entièrement définis par l'instance considérée. De même, la classe `Solution` compose un tableau de `Schedule`, avec un ordonnancement par machine. Chaque `Schedule` n'a de sens que dans le contexte d'une solution donnée : si la solution est supprimée, les ordonnancements associés le sont aussi.

Enfin, des relations de dépendance traduisent l'utilisation ponctuelle de certaines classes par d'autres, sans lien “fort”. La classe `LAHCMetaheuristic` dépend des classes `Heuristic`, `LocalSearch` pour fonctionner car elle utilise une heuristique pour initialiser une solution, applique des opérateurs de recherche locale pour l'améliorer, et va évaluer les solutions générées grâce à l'évaluateur. De son côté, `LocalSearch` dépendra de `Solution` pour accéder aux données et évaluer les effets des opérateurs sur la qualité de la solution.

### c) Agrégation

À l'inverse, certaines relations relèvent de l'agrégation, c'est-à-dire d'une simple référence sans transfert de propriété. Par exemple, la classe `Schedule` agrège une liste de `Job` et référence une `Machine`. Les jobs existent indépendamment de l'ordonnancement : ils appartiennent à l'instance et sont simplement utilisés par les schedules pour définir l'ordre d'exécution sur chaque machine, on pourra réutiliser `Job` plusieurs fois.

### 3. Diagramme UML

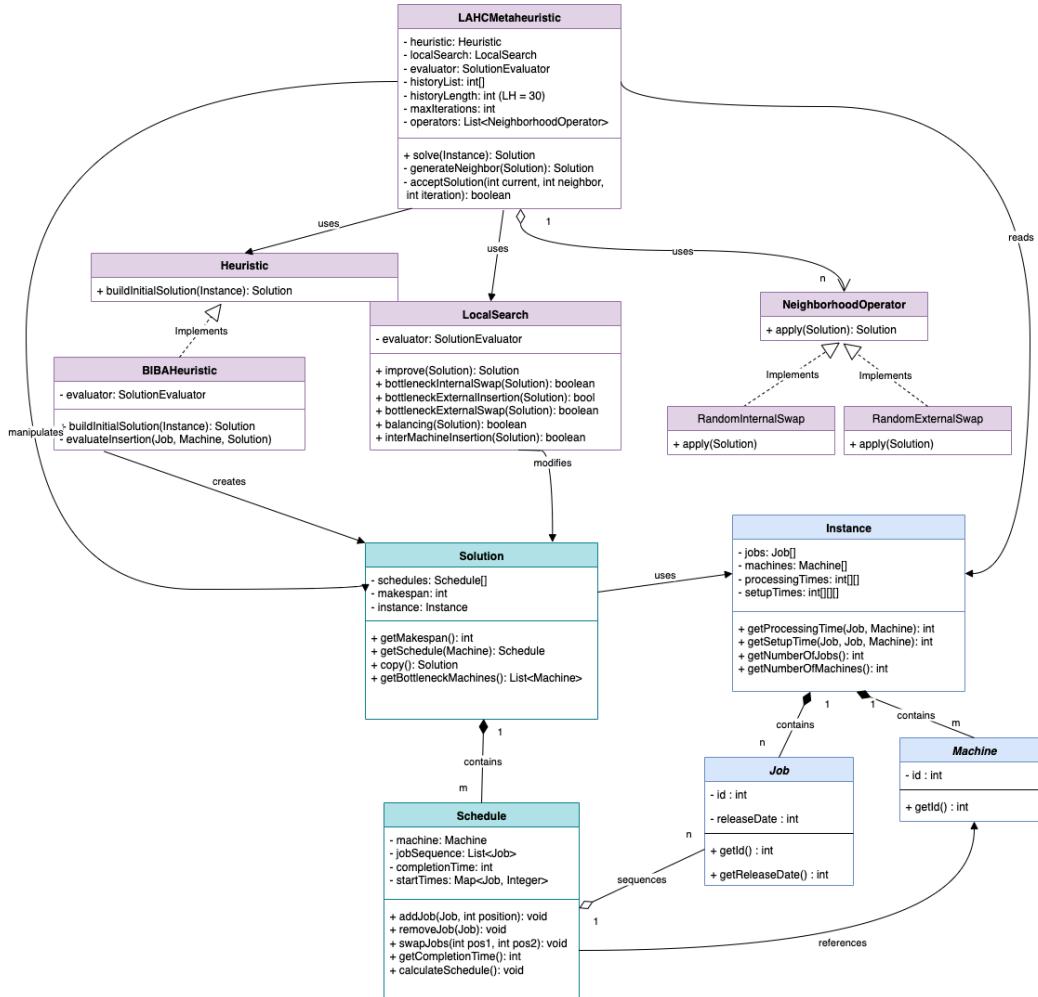


Figure 2: diagramme de classe

Le diagramme est également disponible en suivant le lien suivant pour une meilleure visibilité: [Schéma](#). En violet, on peut retrouver l'ensemble des classes inhérentes à l'implémentation de notre algorithme (ce qui sera nos futures interfaces et les implémentations de ces dernières). De la même manière, les éléments qui seront nécessaires à la solution sont surlignés en turquoise, tandis que les classes “fondamentales”, que nous appelerons “domain” dans le reste du code sont surlignées en bleu. Ces couleurs nous orienterons grandement dans l'architecture du code que nous détaillerons dans les parties ultérieures.

## C. Implémentation en Java

### 1. Structure du projet Java

L'implémentation a été réalisée en Java, en suivant une architecture modulaire inspirée du schéma UML présenté précédemment ainsi que des deux ressources suivantes: [Article Medium](#) et [Clean Architecture](#).

```

LAHC-ParallelMachineScheduling/
├── algo/                      # Algorithmes (heuristiques,
  métahéuristiques, voisinages)
├── domain/                    # Modèle de données
  (Instance, Job, Machine)
├── solution/                  # Représentation des
  solutions (Solution, Schedule)
├── utils/                     # Utilitaires (lecture,
  écriture, visualisation)
└── test/                      # Classes de test
└── resources/                # Fichiers d'instances et
  résultats
└── bin/                       # Fichiers compilés (.class)
└── javafx-sdk-11.0.2/         # Bibliothèque JavaFX
  (standalone)
└── build.sh                   # Script de lancement
  interface graphique
└── tests.sh                   # Script de compilation et
  test console
└── Main.java / MainFX.java   # Points d'entrée

```

Figure 3: Organisation du projet

Les couleurs utilisées dans le diagramme UML sont réutilisées dans l'organisation des packages et des classes. Cela nous permettra en effet de séparer les données du problème de la logique même de l'algorithme ainsi que de laisser la possibilité aux opérateurs d'évoluer sans impacter le code. Le projet est donc structuré en plusieurs packages :

- **domain** : contient les classes liées à la représentation des données “naturelles” du problème (ex. Instance, Job), c'est-à-dire la description des machines et des tâches à planifier.
- **solution** : regroupe les classes qui représentent une solution candidate au problème, notamment Solution (structure globale) et Schedule (ordonnancement pour une machine donnée).
- **algo** : regroupe les algorithmes de recherche de solutions. On y trouve :
  - **algo.neighborhood** : contient les opérateurs de voisinage utilisés pour générer de nouvelles solutions à partir d'une solution existante, et les deux swaps : RandomInternalSwap et RandomExternalSwap.
  - **algo.heuristic** et **algo.metaheuristic**
  - **algo.localSearch** : amélioration de la solution voisine générée
- **test** : contient les classes de tests unitaires et fonctionnels, par exemple *SwapTest*, qui valide le bon comportement des opérateurs de voisinage et des classes principales.

Par ailleurs, aucune bibliothèque externe n'a été nécessaire à ce stade, seuls les packages standards de Java sont utilisés (notamment java.util pour les listes et le random).

## 2. Développement de l'algorithme

Dans cette partie, nous allons “décortiquer” le pseudo-code fourni par l'article, c'est en effet par le biais de celui-ci que nous allons construire brique après brique l'ensemble de notre projet, au delà du simple aspect de classes. Les “éléments de code” correspondent directement aux structures mises en évidence dans le pseudo-code, nous suivrons ainsi la trame suivante pour l'ensemble de nos explications détaillées du code.

```

function LAHC(Instance)
2:   P  $\leftarrow$  0
3:   i  $\leftarrow$  1
4:   best_sol  $\leftarrow$  INIT()
5:   curr_sol  $\leftarrow$  best_sol
6:   while i  $\leq$  Nbiter and P  $\leq$  Non_improv do
7:     | next_sol  $\leftarrow$  NEIGHBOUR(curr_sol)
8:     | next_sol  $\leftarrow$  LOCAL_SEARCH(next_sol)
9:     | if CMAX(curr_sol)  $\leq$  CMAX(next_sol)
10:    | OR CMAX(curr_sol)  $\leq$  H[i%LH] then
11:      | | curr_sol  $\leftarrow$  next_sol
12:    | end if
13:    | if CMAX(curr_sol)  $\leq$  CMAX(best_sol) then
14:      | | best_sol  $\leftarrow$  curr_sol
15:      | | P  $\leftarrow$  0
16:    | end if
17:    | H[i%LH]  $\leftarrow$  CMAX(next_sol)
18:    | i  $\leftarrow$  i + 1
19:    | P  $\leftarrow$  P + 1
20:   end while
21:   return best_sol
end function

```

Figure 4: Pseudo-code papier

### a) Structure des Données “fondamentales” - ①

#### i) La classe INSTANCE

Comme évoqué précédemment, le modèle d'ordonnancement multi-machines est représenté par :

- une Instance, qui contient l'ensemble des jobs à planifier et le nombre de machines disponibles,
- une Solution, qui contient pour chaque machine un Schedule, c'est-à-dire la séquence ordonnée des jobs attribués à cette machine.

La classe **Instance** constitue le cœur de la modélisation de notre problème d'ordonnancement. Elle encapsule toutes les données nécessaires à la description d'une instance du problème, à savoir, l'ensemble des données évoquées dans le **II.A.2**.

```

public class Instance {
    private final Job[] jobs;
    private final Machine[] machines;

    // processingTimes[j][k] = temps de traitement du job j sur la
    machine k (les p_jk)
    private final int[][] processingTimes;

    // setupTimes[i][j][k] = temps de setup entre job i et job j sur
    machine k (les s_ij^k)
    private final int[][][] setupTimes;

    private final int numJobs;
    private final int numMachines;
    //...

```

Figure 5: Classe Instance

Comme nous l'avons vu, nous stockerons donc le temps de traitement du job j sur la machine k dans le tableau de tableau processingTimes[j][k]. De la même manière, setupTimes[i][j][k] stocke le temps de setup pour passer du job i au job j sur la machine k. Par ailleurs, on trouvera ici deux constructeurs:

- Un premier qui initialise les jobs avec une date de disponibilité nulle.
- Un second qui permet de spécifier un tableau de dates de disponibilité

Par exemple, pour l'utiliser, supposons que l'on souhaite créer une instance avec 3 jobs et 2 machines, puis définir les temps de traitement et de setup , on aura alors:

```

Instance instance = new Instance ( 3, 2);
instance.setProcessingTime(0, 0, 5); // Job 0 sur Machine 0 prend 5
unités de temps
instance.setSetupTime(0, 1, 0, 2); // Passer de Job 0 à Job 1 sur
Machine 0 prend 2 unités

```

Figure 6: Exemple utilisation Instance

## ii) Les classes SCHEDULE & SOLUTION

D'une part, on trouve la classe **Schedule**, qui représente l'ordonnancement détaillé d'une machine spécifique. Elle gère la séquence des jobs et calcule les temps de début et de fin de chaque job.

```

//recalculates the schedule (start times, end times, completion time)
public void calculateSchedule() {
    startTimes.clear();
}

```

```

endTimes.clear();

if (jobSequence.isEmpty()) {
    completionTime = 0;
    return;
}

int currentTime = 0;
Job previousJob = null;

for (Job job: jobSequence) {
    //le job ne peut commencer avant sa release date
    int startTime = Math.max(currentTime, job.getReleaseDate());

    //temps de setup
    int setupTime = instance.getSetupTime(previousJob, job,
machine);

    //temps de traitement
    int processingTime = instance.getProcessingTime(job,
machine);

    //enregistrement
    startTimes.put(job, startTime);
    int endTime = startTime + setupTime + processingTime;
    endTimes.put(job, endTime);

    currentTime = endTime;
    previousJob = job;
}

```

Figure 7: Méthode `calculateSchedule()` (classe Schedule)

L'algorithme suit la logique référencée dans l'article pour chaque job  $j$  de la séquence :

- Temps de début =  $\max(\text{temps courant}, r_j)$  (car il faut respecter la release date)
- Temps de setup =  $s_{ijk}$  où  $i$  est le job précédent (ou  $s_{jik}$  si premier job)
- Temps de fin = début + setup +  $p_{jk}$

D'autre part, on trouve la classe **Solution**, qui encapsule l'état de l'affectation des jobs, le calcul du makespan (temps d'achèvement maximal), et fournit des méthodes utilitaires pour manipuler et analyser la solution. Ces attributs principaux sont notamment:

- schedules : tableau de Schedule, représentant la séquence des jobs pour chaque machine.
- instance : Référence à l'instance du problème (ensemble des jobs et machines).
- makespan : Entier stockant le makespan courant de la solution.

- evaluated : booléen qui indique si le makespan est à jour.

Par ailleurs, on pourra également citer certaines méthodes telles que `calculateMakespan()`<sup>3</sup> qui, comme son nom l'indique, calcule le makespan en évaluant chaque machine. En fin de compte, cette classe gère la cohérence du makespan grâce au flag “evaluated”.

### iii) Construction de la Solution Initiale

Conformément à l’Algorithme 1 du papier de référence, nous avons implémenté l’heuristique BIBA pour générer la solution initiale. Cette heuristique constructive gloutonne procède de manière itérative (implémentée dans la méthode `buildInitialSolution()`) :

- ① Partir d'une solution vide (càd aucun job assigné)
- ② Pour chaque job non encore assigné :
  - on évalue l'insertion à la fin de chaque machine
  - on calcule le makespan résultant pour chaque insertion possible
  - puis, on choisit l'insertion qui minimise le makespan
- ③ On répète jusqu'à ce que tous les jobs soient assignés

Figure 8: Principe BIBA (glouton)

## b) Neighbour - ②

### i) L'Opérateur Neighbor

Une fois la solution initiale construite par BIBA, l'algorithme LAHC doit explorer l'espace de solutions en générant des solutions voisines. Une solution voisine est une solution qui diffère légèrement de la solution courante par une modification locale (ici, ce sont nos swap de jobs), permettant ainsi une exploration progressive de l'espace des solutions. L'appel à la fonction de génération de voisins intervient à chaque itération de LAHC (ligne 7 de l’Algorithme 4 du papier), comme illustré ci-dessus.

---

<sup>3</sup> Nous pouvons également parler de la méthode `copy()`, que nous utiliserons dans le reste du projet. Cette dernière permet de manipuler des solutions sans “effet de bord”, c'est-à-dire sans qu'une modification sur l'objet copié ne puisse impacter l'objet original, ou inversement, à cause d'un partage de références internes. Cela arrive souvent lors d'une copie superficielle (aussi appelée shallow copy), dont nous avons pu trouver les détails sur [Dupliquer un objet - cours TSP](#).

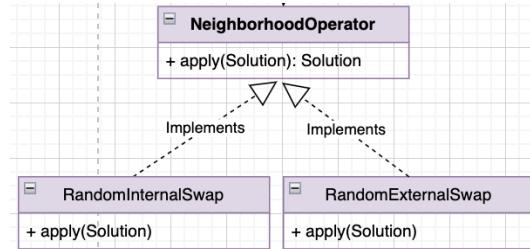


Figure 9: Zoom sur la méthode de génération Neighborhood

L'appel à la fonction `generateNeighbor()` qui instancie l'opérateur de voisinage est défini dans `./algo/neighborhood` et défini le choix entre les deux types de swaps. Ces deux opérateurs seront par ailleurs initialisés dans le constructeur de LAHC.

```

private Solution generateNeighbor(Solution solution) {
    //opérateur probabilité uniforme soit 50%
    NeighborhoodOperator operator =
operators.get(random.nextInt(operators.size()));
    return operator.apply(solution);
}
  
```

Figure 10: GenerateNeighbor

## ii) Les Deux Types de Swaps

L'opérateur `RandomInternalSwap` effectue un échange de deux jobs au sein d'une même machine. Il préserve l'affectation globale des jobs aux machines tout en modifiant l'ordre d'exécution sur une machine. Par exemple, si la machine<sub>o</sub> contient [J<sub>1</sub>, J<sub>2</sub>, J<sub>3</sub>], un swap interne peut produire [J<sub>1</sub>, J<sub>3</sub>, J<sub>2</sub>] ou [J<sub>3</sub>, J<sub>2</sub>, J<sub>1</sub>], etc. De la même manière, l'opérateur `RandomExternalSwap` effectue un échange de jobs entre deux machines différentes. Il modifie quant à lui l'affectation des jobs tout en conservant le nombre total de jobs sur chaque machine. Par exemple, si on considère les machines M<sub>o</sub> = [J<sub>1</sub>, J<sub>2</sub>] et M<sub>1</sub> = [J<sub>3</sub>, J<sub>4</sub>], un swap externe peut donner M<sub>o</sub> = [J<sub>3</sub>, J<sub>2</sub>] et M<sub>1</sub> = [J<sub>1</sub>, J<sub>4</sub>].

En somme, l'internal swap optimise l'ordre d'exécution sur une machine sans modifier la répartition des charges tandis que l'external swap permet de rééquilibrer la charge entre machines et d'échapper à des minima locaux où la répartition actuelle est sous-optimale.

## c) LocalSearch : amélioration par recherche locale - ③

### i) Principe

Après chaque génération de solution voisine, l'algorithme LAHC applique une phase de recherche locale pour améliorer la solution avant de décider de son acceptation (ligne 8 de l'Algorithm 4). Cette étape exploite le voisinage de la solution générée aléatoirement pour trouver rapidement des améliorations locales. La classe `LocalSearch` implémente à cet

effet cinq opérateurs d'amélioration appliqués séquentiellement jusqu'à ce qu'aucune amélioration ne soit trouvée:

```

public Solution improve(Solution solution) {
    Solution improved = solution.copy();
    boolean improvement;

    //on applique les opérateurs jusqu'à "stabilisation"
    do {
        improvement = applyOperators(improved);
    } while (improvement);

    return improved;
}

private boolean applyOperators(Solution solution) {
    boolean improved = false;

    improved |= bottleneckInternalSwap(solution);
    improved |= bottleneckExternalInsertion(solution);
    improved |= bottleneckExternalSwap(solution);
    improved |= balancing(solution);
    improved |= interMachineInsertion(solution);

    return improved;
}

```

Figure 11: méthode `improve()` de la classe LocalSearch

## ii) Les Cinq Opérateurs de Recherche Locale

Contrairement aux opérateurs de voisinage qui sont aléatoires, les opérateurs de recherche locale sont déterministes et orientés: ils ciblent spécifiquement les machines goulot (bottleneck machines), c'est-à-dire les machines dont le temps d'achèvement est égal au makespan. Ils seront appliqués de manière séquentielle et itérative :

- ① Chaque opérateur tente d'améliorer la solution
- ② Si au moins un opérateur trouve une amélioration, tous les opérateurs sont réappliqués
- ③ Le processus s'arrête lorsqu'aucun opérateur ne peut améliorer la solution (point fixe local)

Figure 12: GenerateNeighbor

Cette approche permet en somme d'exploiter le voisinage assez efficacement avant de retourner à la phase d'exploration de LAHC, ce qui accélère la convergence vers des solutions. On pourra donc citer les principes suivants:

- **Opérateur 1 - Bottleneck Internal Swap:** Essaie tous les swaps possibles de deux jobs au sein de chaque machine goulot et conserve celui qui réduit le plus le makespan.
- **Opérateur 2 - Bottleneck External Insertion :** Tente de déplacer un job d'une machine goulot vers une machine non-goulot pour réduire la charge des machines critiques.

Principe : Pour chaque job sur une machine goulot, évalue son insertion à toutes les positions d'une machine non-goulot sélectionnée aléatoirement.

- **Opérateur 3 - Bottleneck External Swap :** Échange un job d'une machine goulot avec un job d'une machine non-goulot dans le but d'obtenir des jobs plus courts (temps de traitement réduit) sur les machines critiques.
- **Opérateur 4 - Balancing :** Équilibre la charge entre machines en déplaçant itérativement le dernier job des machines goulot vers des machines non-goulot, tant que cela améliore ou maintient le makespan.
- **Opérateur 5 : Inter Machine Insertion :** Cherche de meilleures positions pour les jobs en les déplaçant entre machines. Applique l'équation (1) du papier : un mouvement est accepté si la réduction du completion time sur la machine source est supérieure à l'augmentation sur la machine cible, tout en maintenant ou améliorant le makespan. On doit donc vérifier la condition d'acceptation suivante :

$$C_k - C_{k,new} > C_{h,new} - C_h \quad \text{et} \quad C_{max,new} \leq C_{max}$$

ce qui se traduit dans le code par:

```
//on vérifie l'équation (1) du papier
double gainK = currentCk - newCk;
double costH = newCh - currentCh;

if (gainK > costH && newMakespan <= currentMakespan) {
    //mouvement "bénéfique" accepté
}
```

Figure 13: Condition acceptation InterMachineInsertion

#### d) Le Mécanisme d'acceptation de LAHC

Après avoir généré et amélioré une solution voisine, LAHC doit décider si cette nouvelle solution remplace la solution courante (c'est d'ailleurs ici que réside la nouveauté principale du LAHC par rapport au Hill Climbing classique).

```

function LAHC(Instance)
2:   |   P  $\leftarrow$  0
3:   |   i  $\leftarrow$  1
4:   |   best_sol  $\leftarrow$  INIT()
5:   |   curr_sol  $\leftarrow$  best_sol
6:   |   while i  $\leq$  Nbiter and P  $\leq$  Non_improv do
7:   |   |   next_sol  $\leftarrow$  NEIGHBOUR(curr_sol)
8:   |   |   next_sol  $\leftarrow$  LOCAL_SEARCH(next_sol)
9:   |   |   if CMAX(curr_sol)  $\leq$  CMAX(next_sol)
10:  |   |   |   OR CMAX(curr_sol)  $\leq$  H[i%LH] then (4)
11:  |   |   |   |   curr_sol  $\leftarrow$  next_sol
12:  |   |   |   end if
13:  |   |   |   if CMAX(curr_sol)  $\leq$  CMAX(best_sol) then
14:  |   |   |   |   best_sol  $\leftarrow$  curr_sol
15:  |   |   |   |   P  $\leftarrow$  0
16:  |   |   |   end if
17:  |   |   |   H[i%LH]  $\leftarrow$  CMAX(next_sol)
18:  |   |   |   i  $\leftarrow$  i + 1
19:  |   |   |   P  $\leftarrow$  P + 1
20:  |   end while
21:  |   return best_sol
end function

```

Figure 14: Acceptation d'une nouvelle solution

On implémente la double condition d'acceptation via la méthode `acceptSolution()` décrite ligne 9-10 du pseudo-code. Ces conditions se résument de la manière suivante:

- (1) Condition classique (Hill Climbing) :  $\text{Cmax}(\text{neighbor}) \leq \text{Cmax}(\text{current})$ 
  - Si le voisin améliore ou maintient la qualité, on l'accepte toujours
  - C'est le comportement standard d'une recherche locale gloutonne
- (2) Condition de "Late Acceptance" :  $\text{Cmax}(\text{neighbor}) \leq H[i \% LH]$ 
  - On compare le voisin avec la solution trouvée il y a LH itérations
  - Permet d'accepter des solutions qui dégradent temporairement la qualité courante

Figure 15: Fonctionnement mécanisme acceptation

Par exemple, regardons pour 100 itérations, un `currentMakespan = 150` et `neighborMakespan = 155` (pire que current). On a donc le makespan à l'itération 70:

$$H[100\%30] = H[10] = 160$$

Ainsi, on a  $155 \leq 160 \Rightarrow \text{TRUE}$ , on accepte le voisin, même si  $155 > 150$ , on accepte car on progresse par rapport à l'itération 70.

### e) Mise à jour de la meilleure solution

Parallèlement au critère d'acceptation, LAHC maintient la meilleure solution globale rencontrée durant toute la recherche (lignes 13-16 du pseudo-code) :

```

function LAHC(Instance)
2:   P  $\leftarrow$  0
3:   i  $\leftarrow$  1
4:   best_sol  $\leftarrow$  INIT()
5:   curr_sol  $\leftarrow$  best_sol
6:   while i  $\leq$  Nb_iter and P  $\leq$  Non_improv do
7:     | next_sol  $\leftarrow$  NEIGHBOUR(curr_sol)
8:     | next_sol  $\leftarrow$  LOCAL_SEARCH(next_sol)
9:     | if CMAX(curr_sol)  $\leq$  CMAX(next_sol)
10:    | OR CMAX(curr_sol)  $\leq$  H[i%LH] then
11:      | | curr_sol  $\leftarrow$  next_sol
12:    | end if
13:    | if CMAX(curr_sol)  $\leq$  CMAX(best_sol) then
14:      | | best_sol  $\leftarrow$  curr_sol
15:      | | P  $\leftarrow$  0
16:    | end if
17:    | H[i%LH]  $\leftarrow$  CMAX(next_sol)
18:    | i  $\leftarrow$  i + 1
19:    | P  $\leftarrow$  P + 1
20:   end while
21:   return best_sol
end function

```

Figure 16: Suivi de la meilleure solution

```

// maj meilleure solution
if (neighborCost <
bestSolution.getMakespan()) {
  bestSolution = neighbor.copy();
  lastImprovementIteration =
iterationCount;
  nonImprovementCount = 0; //reinit
count

if (iterationCount % 100 == 0) {
  System.out.printf("Iteration %d:
New best makespan = %d%n",
iterationCount,
neighborCost);
}
} else {
  nonImprovementCount++;
// P  $\leftarrow$  P + 1
(1..20)
}

```

En somme, bien que **currentSolution** puisse se dégrader temporairement (exploration), **bestSolution** restera monotone, c'est-à-dire, elle ne peut que s'améliorer ou rester constante.

### f) Gestion de la liste H

Comme nous avons pu le voir, la liste "H" est le mécanisme central de LAHC. C'est elle qui stocke les makespans des solutions rencontrées pour permettre la comparaison "late acceptance".

```

//maj liste historique (ligne 17)
historyList[iterationCount % historyLength] = neighborCost;

//itération (ligne 18)
iterationCount++;

```

Cette dernière a un fonctionnement circulaire : la liste a une taille fixe LH (30 selon le papier après tuning), l'index circulaire *i* % LH permet de réutiliser les emplacements, puis, à chaque itération, on écrase la valeur la plus ancienne. Prenons un exemple pour une liste plus petite, comme LH = 5 :

```

Itération 0: H = [100, 100, 100, 100, 100] (init)
Itération 1: H = [100, 95, 100, 100, 100] (màj H[1])
Itération 2: H = [100, 95, 98, 100, 100] (màj H[2])

```

```

...
Itération 5: H = [97, 95, 98, 100, 100]      (H[0] réécrasé)
Itération 6: H = [97, 93, 98, 100, 100]      (H[1] réécrasé)

```

### g) Condition d'arrêt

#### i) Arrêt

Enfin, LAHC s'arrête lorsque l'une des deux conditions est satisfaite (ligne 6 du pseudo-code, ou boucle while dans le code) :

- on a atteint le nombre maximum d'itérations : iterationCount  $\geq$  maxIterations
- ou la limite de non-amélioration : nonImprovementCount  $\geq$  nonImprovementLimit

En somme, Le premier critère évite une exécution infinie, tandis que le second permet un arrêt anticipé si la convergence est atteinte. Selon le papier, LAHC trouve sa meilleure solution dans les premiers 5% du temps d'exécution.

#### ii) Paramètres

Nous venons d'évoquer la condition d'arrêt utilisant le paramètre `nonImprovementCount`. Bien que cette variable apparaisse dans le pseudo-code de l'algorithme (ligne 6), aucune valeur n'est spécifiée dans l'article. Nous avons donc fixé arbitrairement `nonImprovementCount` à 1000 dans notre implémentation (seuil de stagnation au-delà duquel la recherche sera considérée comme ayant convergé).

Selon l'article, les auteurs ont utilisé l'outil ParamILS pour optimiser les paramètres de LAHC, dont le seul paramètre à optimiser est la longueur de la listeLH, testée dans l'intervalle {10, 20, 30, 40, 50}. ParamILS a déterminé que LH = 30 offrait les meilleures performances, valeur que nous avons donc adoptée dans notre implémentation. Le pseudo-code mentionne également une variable `Nb_iter` (dont la valeur est supposément fixée à 20 itérations dans le papier) comme nombre maximum d'itérations, mais l'article précise en réalité que les auteurs ont utilisé un critère de temps plutôt qu'un critère d'itérations, défini par :

$$Time_{limit} = \frac{n * m}{2} \text{ secondes}$$

où  $n$  est le nombre de jobs et  $m$  le nombre de machines (permet de garantir un temps d'exécution proportionnel à la taille de l'instance).

## 3. Tests et validation

### a) test/ : Tests unitaires

Avant d'implémenter la logique principale de notre programme dans Main.java, nous avons mis en place une série de tests unitaires dans le dossier `test/`.

Ces tests ont permis de vérifier individuellement le bon fonctionnement des différentes briques de notre projet :

- Heuristiques (BibaTest.java) : vérifie que la construction initiale de solution via l'heuristique BIBA produit des solutions valides et cohérentes.
- Opérateurs de voisinage (SwapTest.java) : teste les fonctions de swap interne et externe pour s'assurer que les échanges sont bien appliqués et que la structure de la solution reste valide.
- Recherche locale (LocalSearchTest.java) : contrôle que l'algorithme d'amélioration locale diminue effectivement le makespan ou le maintient constant sans générer de solutions invalides.
- LAHC (LAHCTest.java) : teste l'intégration de la météahéuristic Late Acceptance Hill Climbing sur des petites instances pour vérifier la convergence et la stabilité du processus.

Grâce à ces tests, nous avons pu détecter et corriger des erreurs de manipulation de séquences de jobs, avant de passer à l'intégration complète. Ces tests sont exécutables grâce aux commandes présentes dans le fichier tests.sh :

```
chmod +x build-test.sh  
./build-test.sh
```

### b) Lecture / écriture des instances d'un fichier

Pour faciliter la manipulation des instances de test, nous avons également exploité la possibilité de lire un fichier txt (permettant de stocker toutes les données nécessaires du problème) dans la classe **InstanceReader** avec la forme suivante:

**Ligne 1 :** n m (nombre de jobs et machines)

**Ligne 2 :**  $r_0 \ r_1 \ \dots \ r_{n-1}$  (release dates)

**Lignes 3-n+2 :** matrice n\*m des temps de traitement ( $p_{jk}$ )

**Lignes suivantes :** m matrices n\*n des temps de setup ( $s_{ijk}$  pour chaque machine)

### c) Main.java

Une fois chaque composant validé séparément, nous avons intégré l'ensemble dans la classe Main.java. Ce fichier constitue le point d'entrée du programme :

- Il crée une instance de test avec ses temps de traitement et de setup
- Il configure et lance la météahéuristic LAHC avec l'heuristique BIBA comme point de départ
- Il mesure le temps d'exécution et affiche les résultats finaux : makespan obtenu, séquences de jobs pour chaque machine, et statistiques de performance.

#### d) Problèmes rencontrés

##### i) LocalSearch

Lors des premières exécutions, nous avons constaté que le programme devenait très lent après un certain nombre d'itérations, malgré la contrainte de temps imposée dans la boucle LAHC. Ce ralentissement (voire blocage) provient principalement de la méthode `LocalSearch.improve()`, appelée à chaque itération. Celle-ci effectue une descente locale complète jusqu'à un optimum, en appliquant successivement cinq opérateurs tant qu'ils apportent des améliorations. Or, chaque opérateur explore un grand nombre de mouvements possibles (swaps internes ou externes, insertions entre machines, etc.) et recalcule le makespan à chaque tentative, ce qui rend cette étape très coûteuse, typiquement  $O(n^2)$  ou plus par opérateur. Ainsi, une seule itération peut prendre plusieurs secondes, empêchant le programme de vérifier régulièrement la borne de temps et entraînant un dépassement significatif du temps prévu.

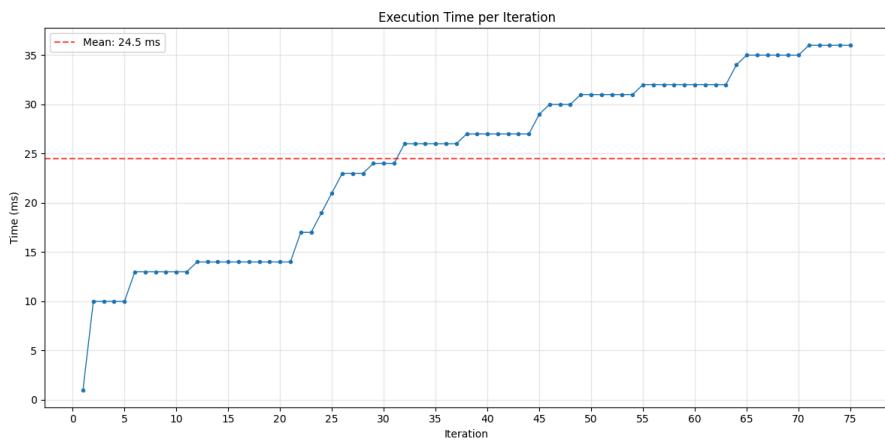


Figure 17: Si le nombre d'itérations est trop important, le temps d'exécution explose

Pour résoudre cela, nous avons cherché plusieurs solutions:

- Réduire la fréquence d'appel de la recherche locale, par exemple, l'appliquer seulement toutes les 10 itérations, ou avec une certaine probabilité (ex. 10 %), qui permet de conserver la capacité d'amélioration sans alourdir chaque itération. Nous avons testé cette possibilité (l. 97-99 de `LAHCMetaheuristic.java`) et avons conclus que cette solution fonctionnait et n'avait que peu d'impact sur le résultat.
- Appliquer la recherche locale uniquement aux solutions acceptées plutôt que de l'appliquer systématiquement sur tous les voisins générés, on peut d'abord tester le critère d'acceptation LAHC, puis n'appliquer `improve()` que sur les solutions qui sont effectivement retenues pour éviter de « gaspiller » des calculs sur des solutions rejetées;
- Limiter la profondeur de la recherche local, c'est-à-dire au lieu de boucler jusqu'à convergence, on peut n'appliquer les cinq opérateurs qu'un certain nombre de fois par appel. C'est l'option vers laquelle nous nous sommes dirigés;

### ii) Liste historique

Nous avons également corrigé une erreur conceptuelle : initialement, la liste historique était mise à jour avec le coût du voisin (neighborCost) même lorsque celui-ci n'était pas accepté et donc faussait le mécanisme d'acceptation, qui repose justement sur la comparaison entre le coût actuel et une valeur antérieure du chemin de recherche. Nous avons remplacé cette instruction par :

```
historyList[iterationCount % historyLength] =  
currentSolution.getMakespan();
```

### iii) Temps de setups

Lorsque nous avons implémenté l'interface graphique, nous nous sommes rendus compte que notre code autorisait les setups à commencer avant la release date, puis chechait la release date pour le temps de process. Encore une fois, il s'agissait d'une erreur conceptuelle que nous avons corrigé.

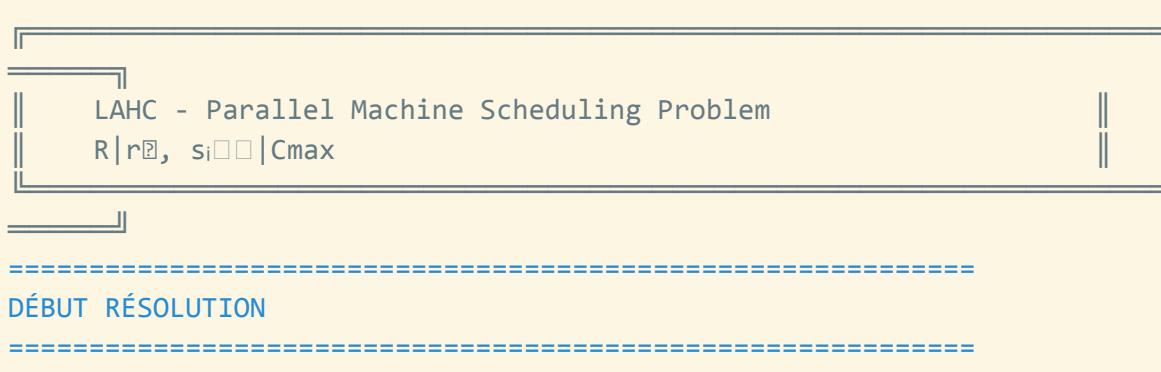
Enfin, pour éviter les surcharges de calcul inutiles, nous avons ajouté une condition d'arrêt dès qu'un certain nombre d'itérations est atteint sans amélioration (par exemple 1000). Cela permet de préserver la réactivité du programme sur les petites instances.

## e) Résultat

Ainsi, avec une instance random créée grâce à la méthode `createRandomInstance()` de la classe `InstanceReader`:

```
//./Main.java  
Instance instance = InstanceReader.createRandomInstance(5, 2, 10, 5,  
0.5);
```

On obtient dans le terminal :



The terminal window displays the following text:

```
LAHC - Parallel Machine Scheduling Problem  
R|r[], si[]|Cmax  
=====
```

At the bottom of the terminal window, there is a blue banner with the text "DÉBUT RÉSOLUTION" in white, followed by another blue banner at the bottom.

```
Generating initial solution with BIBA heuristic...
Initial makespan: 23
LAHC with 5 max iterations...
Iteration 1: 0 ms
Iteration 2: 12 ms
Iteration 3: 12 ms
...
LAHC finished:
Total iterations: 22
Total time: 0.016 s
Last improvement at iteration: 2
Initial makespan: 23
Final makespan: 21
Improvement: 8,70%
```

#### SOLUTION OBTENUE:

Makespan (Cmax): 21

---

#### STATISTIQUES

---

Temps d'exécution: 54 ms  
Jobs: 5  
Machines: 2

---

#### DIAGRAMME DE GANTT

---

Légende: █=s<sub>ijk</sub> / █=p<sub>ij</sub> | ·=r<sub>j</sub>

M0: ...█[J4:████] █[J1:███████] █[J0:████] (C=21) ← BOTTLENECK  
M1: .....█[J3:████] █[J2:██████] (C=20)

Makespan (Cmax) = 21



## D. Validation et Analyse des Résultats

## 1. Jeux de données utilisés (benchmarks)

Les résultats de ces benchmarks sont retrouvables dans le répertoire ./resources/out/. Dans l'ensemble de cette étude, nous fixerons seulement les trois paramètres suivants afin de ne pas avoir à manipuler des données trop disparates.

- max processing time : 10
- max setup time : 5
- release date factor : 0.5

Nous jouerons ensuite sur le nombre de machines et/ou de jobs selon les benchmarks.

```
Instance instance = InstanceReader.createRandomInstance(randomJobNumber,  
randomMachineNumber, 10, 5, 0.5);
```

### a) Premier benchmark: 100 “petites instances”

Dans ce premier test, nous nous sommes concentrés, comme proposé dans le papier sur les petites instances (c'est-à-dire des instances aléatoires comprenant entre 5 et 20 jobs et entre 2 et 5 machines). Plus précisément, notre dataset est composé de:

<b>Nombre total d'exécutions:</b> 100
<b>Jobs:</b> min=5, max=20, moyenne=13.0
<b>Machines:</b> min=2, max=5, moyenne=3.4

Sur l'ensemble des 100 instances testées, LAHC n'améliore pas systématiquement la solution initiale fournie par BIBA (seulement dans 74% des cas) mais ces résultats montrent que la métaheuristique est efficace sur ce type de problème. Concernant l'amélioration du makespan, on a:

Initial moyen: 31.7
Final moyen: 28.5
Amélioration moyenne: 8.85%
Amélioration médiane: 8.00%
Amélioration max: 29.00%

L'amélioration médiane, très proche de la moyenne suggère une distribution relativement homogène des performances. En outre, la variabilité des améliorations (écart-type) indique que certaines instances bénéficient plus que d'autres de l'optimisation locale.

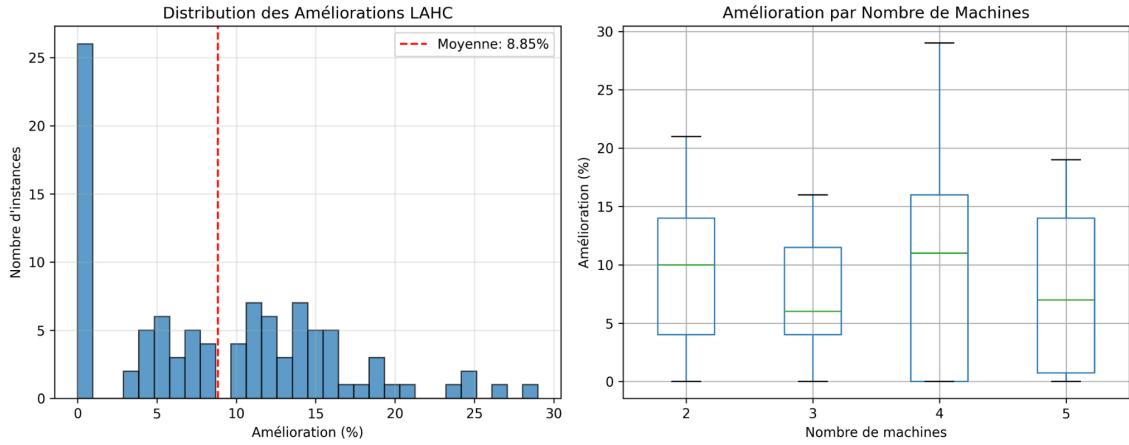


Figure 18: Amélioration selon instances

On peut également s'amuser avec nos données et créer heatmap qui va nous permettre de visualiser comment l'amélioration varie selon le nombre de jobs et de machines.

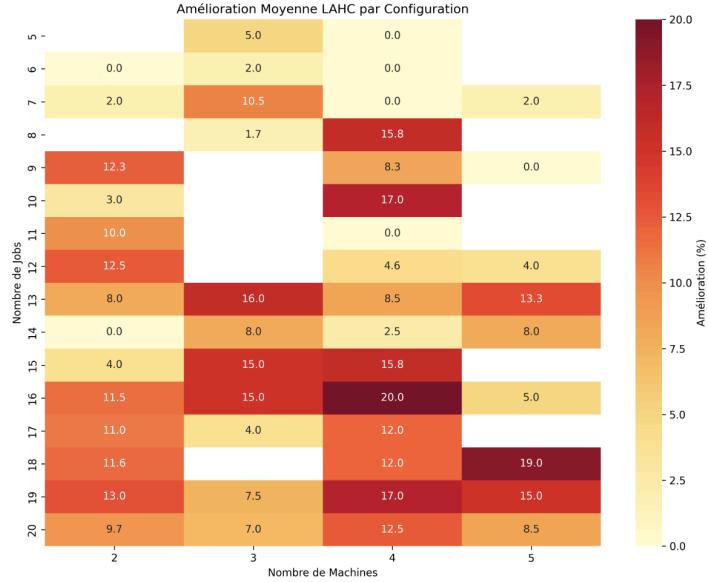


Figure 19: Heatmap Instances

## Performances:

Itérations moyennes: 32  
 Temps d'exécution moyen: 223 ms  
 Temps d'exécution médian: 79 ms

Enfin, on remarque que le nombre d'itération ne joue nullement sur le facteur d'amélioration, ce qui conforte nos choix en matière de développement.

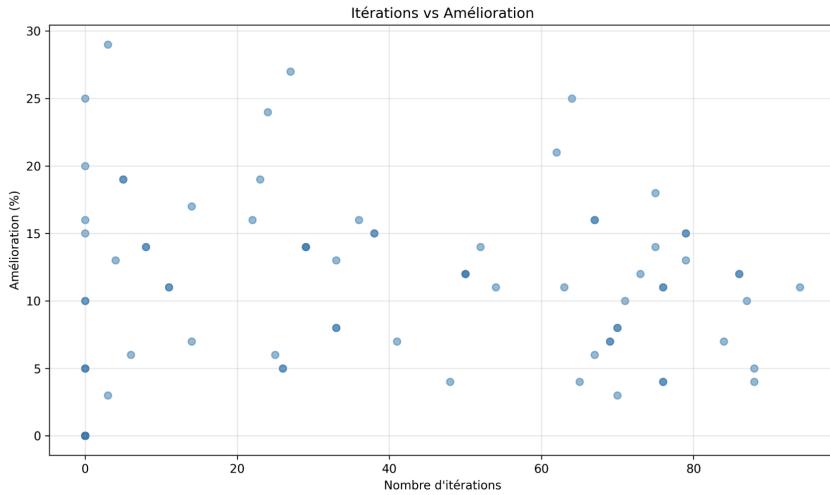


Figure 20: Évolution du % d'amélioration du makespan en fonction du nombre d'itérations

### b) Second benchmark: 100 instances moyennes

Pour ce second benchmark, on regarde des instances comportant entre 20 et 40 jobs répartis entre 2 et 8 machines. On remarque immédiatement que les traitements sont bien plus longs. Nous étudierons le benchmark dont la composition est la suivante:

Nombre total d'instances: 100 Jobs: min=20, max=40, moyenne=30.7 Machines: min=2, max=8, moyenne=4.9
--

#### Amélioration du makespan:

Initial moyen: 46.7 Final moyen: 42.6 Amélioration moyenne: 7.50% Amélioration médiane: 8.00% Amélioration max: 27.00%
--

À nouveau, on obtient que seuls 75% des instances ont pu être améliorées, très similaire à ce que l'on a obtenu précédemment. On retrouve également des résultats similaires en terme d'amélioration avec les plus petites instances, mais pour un temps d'exécution pratiquement deux fois plus important en moyenne.

#### Performances:

Itérations moyennes: 34 Temps d'exécution moyen: 444 ms Temps d'exécution médian: 175 ms
--

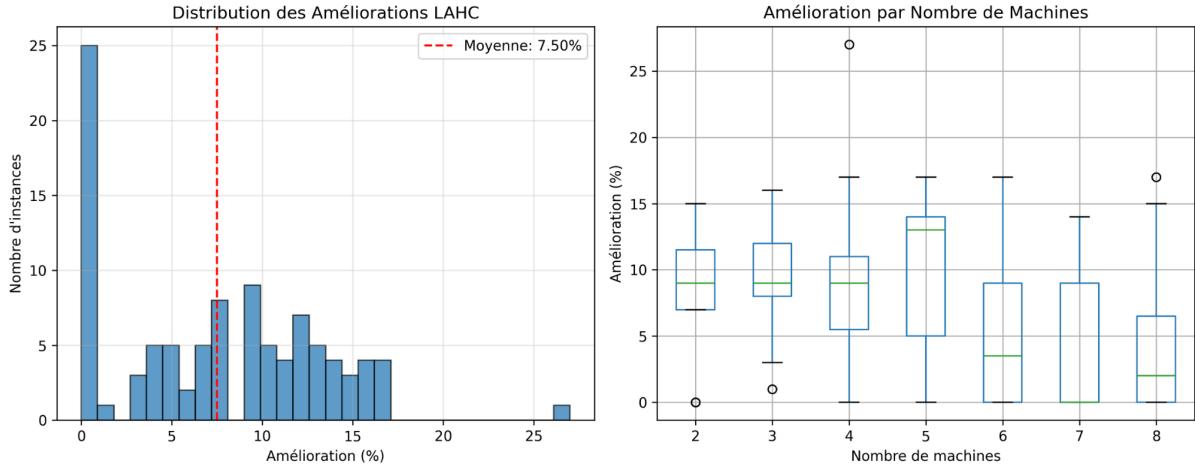


Figure 21: Distribution améliorations LAHC pour Instances moyennes

## 2. Analyse critique

### a) Efficacité et performance

**Mécanisme d'acceptation efficace :** Le principe d'acceptation retardée permet à LAHC d'échapper aux optima locaux car une solution n'est acceptée que si elle est meilleure que celle observée L itérations auparavant, même si elle dégrade la solution courante. Ce mécanisme offre un bon compromis entre exploration et exploitation sans nécessiter de paramètre de température.

**Amélioration systématique :** Sur l'ensemble de nos tests, LAHC améliore 74% des solutions initiales générées par BIBA.

**Temps d'exécution raisonnables :** Pour les instances de taille modérée (5-20 jobs, 2-5 machines), les temps d'exécution restent inférieurs à quelques secondes, ce qui permet une utilisation assez efficace.

### b) Limites et pistes d'amélioration

Malgré ces points forts, nous avons pu observer plusieurs limitations lors des tests et de l'analyse du code.

**Dépendance à la solution initiale :** La qualité finale de la solution dépend fortement de la solution de départ fournie par BIBA. Si BIBA génère une solution de mauvaise qualité (par exemple sur des instances avec de nombreuses contraintes de release dates), LAHC peut avoir du mal à s'en écarter suffisamment. Une solution initiale médiocre limite le potentiel d'amélioration.

**Scalabilité sur grandes instances :** Bien que non testées dans le cadre de ce projet, des instances de très grande taille (50+ jobs, 10+ machines) nécessiteraient probablement un

ajustement des paramètres (augmentation de L, critère d'arrêt adaptatif) ou une modification de la stratégie d'exploration pour maintenir un temps de calcul raisonnable.

**Pas de diversification :** LAHC explore intensivement autour de la solution courante mais ne dispose pas de mécanisme de redémarrage ou de diversification pour sortir de bassins d'attraction. Sur des instances avec de nombreux optima locaux, l'algorithme peut stagner, comme nous avons pu le constater.

**Restriction nombre itération:** Pour éviter que le programme tourne à l'infini, nous avons dû modifier le critère d'arrêt de LAHC avec un critère de temps ainsi que de restreindre le nombre d'itérations du LocalSearch à 100.



### III. Développement d'une Interface Graphique

Cette interface devait nous permettre de générer des instances, lancer l'exécution de l'algorithme et visualiser les résultats sous une forme à la fois textuelle et graphique (notamment via un diagramme de Gantt).

#### A. Conception de l'Interface

Pour développer cette interface, nous avons choisi d'utiliser la bibliothèque JavaFX, qui nous semblait être la plus simple (séparation entre la logique et la “présentation”). Par ailleurs, l'interface graphique suit une architecture événementielle plutôt qu'une architecture MVC (Modèle-Vue-Contrôleur) [cf [Annexe A3](#)], ce qui s'explique par la nature relativement simple de l'application. Nous conserverons ainsi toute la logique de calcul (heuristiques, voisinages, recherche locale, etc.) dans les packages `algo/`, `domain/`, et `solution/`. De même, on aura les deux composants principaux :

- `MainFX.java` : Point d'entrée de l'application, gère l'interface utilisateur et les interactions
- `GanttFX.java` : Utilitaire de visualisation dédié à la génération des diagrammes de Gantt, dans `utils/`

##### a) Main

Les événements sont gérés par des lambda expressions qui appellent directement les méthodes appropriées, on définit les choix possibles dans le switch suivant:

```
switch (inputType) {  
    case "Depuis un fichier":  
        instance =
```

```

InstanceReader.readFromFile("resources/Instance.txt"); //TODO : file
chooser maybe ?
        break;
    case "Entrée manuelle":
        outputArea.appendText("Mode manuel non encore
implémenté.\n");
        return;
    default: //random instance
        int randomJobNumber = 5 + (int)(Math.random() * 6);
// 5 à 10 jobs
        int randomMachineNumber = 2 + (int)(Math.random() *
3); // 2 à 4 machines
        instance =
InstanceReader.createRandomInstance(randomJobNumber,
randomMachineNumber, 10, 5, 0.5);
    }
}

```

### b) Gantt

**Note de transparence :** La classe `GanttFX.java` a été développée avec l'assistance de Claude AI. Face à la complexité du rendu graphique avec JavaFX, j'ai sollicité l'aide de l'IA pour générer une première version fonctionnelle de la visualisation, avant de le débugger et de compléter à la main.

En terme de fonctionnement, la logique de cette classe reproduit exactement le calcul fait dans `Schedule.calculateSchedule()`.

## B. Résultat

L'interface permet :

- Sélection du type d'instance : aléatoire, depuis un fichier, ou manuelle (non implémentée)
- Lancement de la résolution : Exécution de l'algorithme LAHC via un bouton
- Affichage des résultats : Zone de texte pour les logs et informations de la solution
- Visualisation Gantt : Bouton pour ouvrir le diagramme de Gantt dans une fenêtre séparée

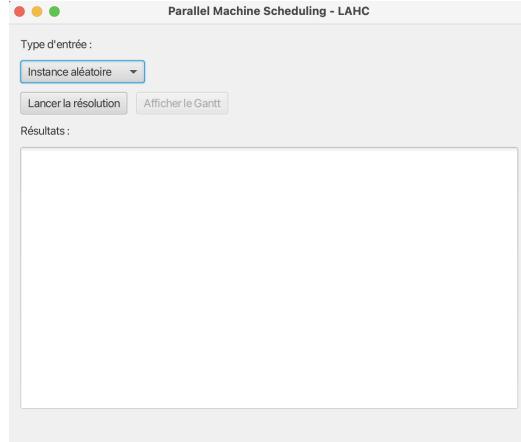


Figure 20: Menu MainFX

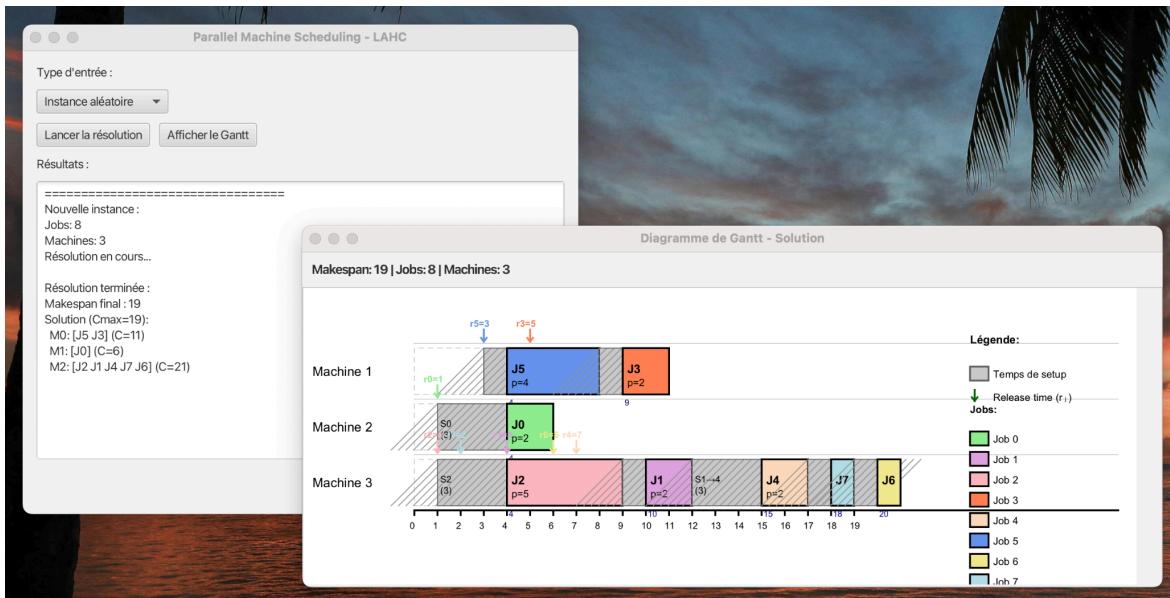


Figure 21: Interface graphique

## Conclusion

Ce projet m'a permis de concevoir et d'implémenter intégralement un solveur pour le problème d'ordonnancement  $R \mid r_j, s_{ijk} \mid C_{max}$ , depuis la modélisation objet jusqu'à la visualisation graphique des solutions. J'y ai développé les heuristiques BIBA et LAHC tout en respectant les contraintes formelles du problème (release dates, setup times), et j'ai réalisé une interface graphique fonctionnelle sous JavaFX.

Nous avons initialement eu des difficultés sur la compréhension fine du mécanisme d'acceptation de LAHC (différence subtile entre "accepter une solution moins bonne que la courante" et "accepter une solution meilleure qu'une solution historique x itérations

auparavant") et sur la validation de l'implémentation. Au fur et à mesure du développement, les difficultés rencontrées se sont déplacées vers une part le debugging des opérateurs de voisinage (les swaps entre machines nécessitaient une bonne gestion de l'invalidation et du recalculation des schedules affectés pour éviter des incohérences), et d'autre part la maîtrise de JavaFX Canvas pour la visualisation Gantt, dont la complexité a justifié le recours à une IA suivie d'une phase de debugging assez intensive et d'adaptation au modèle de données existant.

En somme, ce projet a renforcé mes compétences en algorithmique (modélisation, conception d'opérateurs de voisinage, mise en œuvre de métaheuristiques), en programmation orientée objet (structuration du code, manipulation de collections) et en validation (debugging précis, analyse expérimentale). La confrontation entre théorie et pratique m'a également montré que les articles scientifiques donnent l'essence d'un algorithme, mais que son implémentation exige de nombreuses décisions de conception influant sur les performances. Pour la suite, nous pourrions envisager plusieurs pistes d'amélioration comme enrichir l'interface graphique, élargir les opérateurs de voisinage, automatiser les benchmarks ou encore explorer des extensions plus avancées comme l'hybridation algorithmique.

Au final, ce projet a été une expérience complète et formatrice, alliant théorie, pratique et créativité, et m'a permis d'acquérir des compétences solides en optimisation, en programmation et en conception logicielle.

---

## Bibliographie

- ❖ Hudry, A. (s. d.). Dupliquer un tableau, clonage. Télécom Paris. Consulté le 1 octobre 2023 sur <https://perso.telecom-paristech.fr/hudry/coursJava/avance/dupliquer.html>
- ❖ GeeksforGeeks. (s. d.). Complete Guide to Clean Architecture. Consulté le 5 octobre 2023 sur <https://www.geeksforgeeks.org/system-design/complete-guide-to-clean-architecture/>
- ❖ Kelly, R. (2023, septembre 19). Programming Fundamentals Part 5: Separation of Concerns & Software Architecture. Medium. <https://rkay301.medium.com/programming-fundamentals-part-5-separation-of-concerns-software-architecture-fo4a900a7c5>
- ❖ Hertz, A. (s. d.). Métaheuristiques. École Polytechnique de Montréal, GERAD. Consulté le 13 octobre 2023 sur <https://www.gerad.ca/~alainh/Metaheuristiques.pdf>
- ❖ Darwin, I. F. (2020). *Java Cookbook: Problems and Solutions for Java Developers* (4th ed.). O'Reilly Media.

- ❖ OptaPlanner Documentation 6.2 - Chapter 10: Local Search\*, Red Hat, 2015. [En ligne]. <https://docs.optaplanner.org/6.2.0.Final/optaplanner-docs/html/ch10.html>.
- ❖ M. K. Schäfer, "JavaFX Tutorial - Introduction," Code.makery, 2022. [En ligne]. Disponible: <https://code.makery.ch/fr/library/javafx-tutorial/part1/>
- ❖ F. Michel, "Introduction à JavaFX," LIRMM, Université de Montpellier, 2019. [En ligne]. Disponible: <https://www.lirmm.fr/~fmichel/ens/java/cours/JavaFX.pdf>
- ❖ "JavaFX Documentation - Installation," OpenJFX, 2023. [En ligne]. Disponible: <https://openjfx.io/openjfx-docs/#install-javafx>
- ❖ "Why does JavaFX not work under OpenJDK 17 from Homebrew?", Stack Overflow, Nov. 2021. [En ligne] Disponible: <https://stackoverflow.com/questions/69965469/why-does-javafx-not-work-under-openjdk-17-from-homebrew>
- ❖ J.-M. Doudoux, "Les expressions lambda en Java," JM Doudoux, 2023. [En ligne]. Disponible: <https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm>



## Annexes

### A1 - Exemple : un autre problème d'ordonnancement

Pour mieux comprendre notre problème, nous pourrons étudier l'exemple suivant, fortement similaire à celui présenté dans l'article. On prendra le cas de 3 jobs et 2 machines.

Job	$r_j$	$p_{j1}$	$p_{j2}$
1	0	3	5
2	2	2	4
3	1	4	1

Tableau 1 :  $r_j$  et  $p_{jk}$  pour 3 jobs et 2 machines

De même, on prendra les valeurs suivantes pour les temps de setup des deux machines. On a donc à titre d'exemple le cas du job 3 qui démarre après le job 2 sur la machine 1, son temps de setup  $s_{231} = 6$ .

$s_{ij1}$	1	2	3		$s_{ij2}$	1	2	3
1	2	5	3		1	3	4	5
2	4	1	6		2	1	2	4
3	7	2	2		3	2	6	1

Tableaux 2 & 3 : temps de setups des machines 1 à 3

Une solution que l'on pourrait proposer à cet exemple est donc représentable dans le schéma suivant:

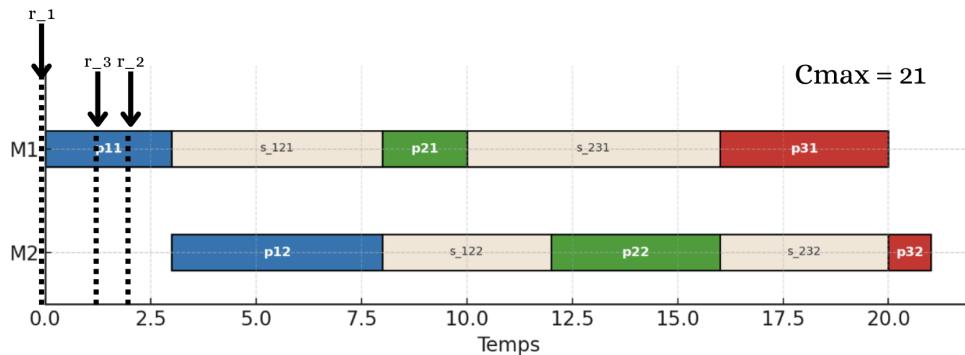


Figure 1: Gantt, instance 3 jobs, 2 machines (ordre (1 2 3), Cmax = 21)

## A2 - Synthèse de l'article

On peut dresser le tableau synthétique suivant qui résume l'ensemble des points évoqués ci-dessus:

Aspect	Description
Problème étudié	$R   r_j, s_{ijk}   C_{max}$ - Premier travail combinant release dates ET setup times machine-dépendants
Contribution principale	Benchmark de 1620 instances (3 sets : small, medium, large) publiquement disponible
Méthodes proposées	3 heuristiques (BIBA, GRASP, Meta-RaPS) + 3 métahéuristiques (LAHC, SA-v1, SA-v2)
Meilleure heuristique	BIBA (Best Insertion Based Approach) sur tous les sets d'instances
Opérateurs utilisés	2 voisnages (internal/external swap) + 5 recherches locales intensives
Résultats	Aucune métahéuristiche ne domine : LAHC (small), SA-v1 (medium), SA-v2 (large)

Avantage de LAHC	Plus simple (1 seul paramètre $L_h$ ), converge rapidement (<5% du temps), RPD=0.05% sur small
Paramétrage	Optimisation automatique via ParamILS ( $L_h=30$ pour LAHC)
Conclusion	Les caractéristiques des instances (taille, nombre de machines) impactent fortement le choix de la métaheuristique

### A3 - Pourquoi pas MVC ?

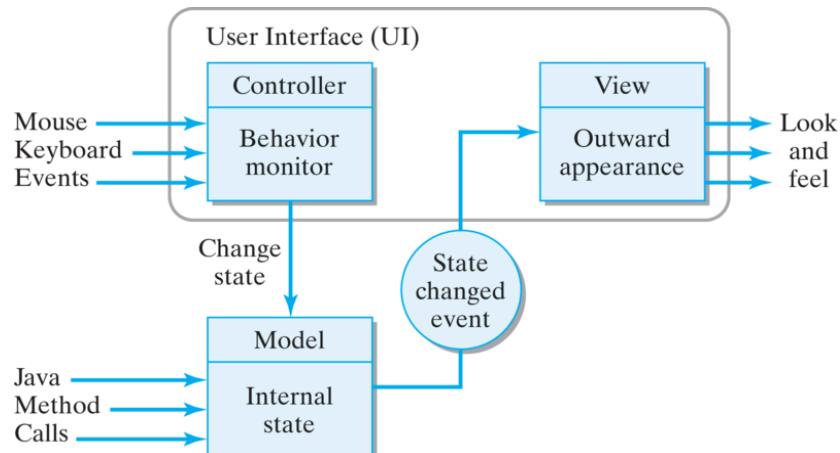


Figure 2: Schéma architecture MVC

Dans une architecture MVC traditionnelle, on aurait :

- Modèle : Les classes de données (Instance, Solution, Schedule)
- Vue : L'interface graphique (MainFX, GanttFX)
- Contrôleur : Logique de gestion des événements

Cependant, JavaFX intègre naturellement la logique de contrôle dans les composants de vue via les listeners et les event handlers, ce qui est plus direct pour une application de cette taille et évite la complexité d'une séparation MVC formelle qui serait excessive ici.

#### A4 - La taille de l'instance joue-t-elle sur le temps d'exécution ?

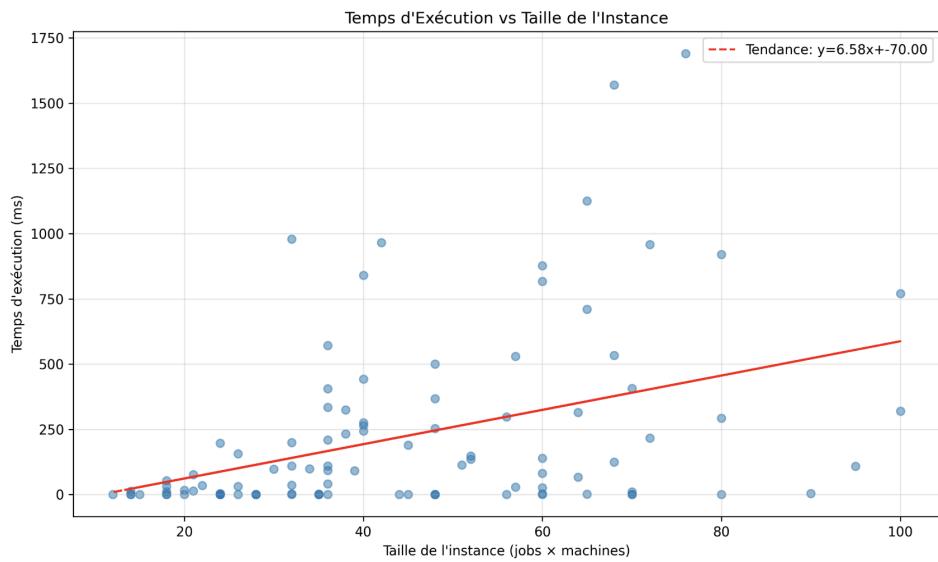


Figure 3: Évolution du temps d'exécution en fonction de la taille de l'instance (testé sur de petites instances entre 5 et 20 jobs)

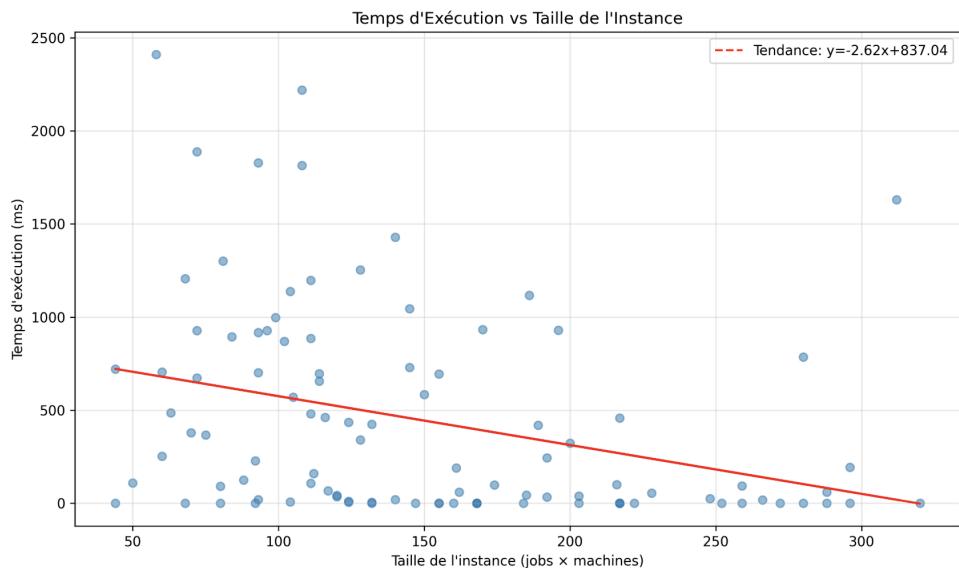


Figure 4: Évolution du temps d'exécution en fonction de la taille de l'instance (testé sur des moyennes instances entre 20 et 40 jobs )