Min Makespan

Table des matières

[Introduction 2](#_Toc29995181)

[Résumé du sujet 2](#_Toc29995182)

[Algorithmes 2](#_Toc29995183)

[Équipe 2](#_Toc29995184)

[Le programme 3](#_Toc29995185)

[Conception 3](#_Toc29995186)

[Interface 3](#_Toc29995187)

[Structure de données 3](#_Toc29995188)

[Usage 3](#_Toc29995189)

[Algorithmes implémentés 4](#_Toc29995190)

[Résultats du programme 4](#_Toc29995191)

[Discussion sur le programme 6](#_Toc29995192)

[Complexité en temps 6](#_Toc29995193)

# Introduction

## Résumé du sujet

Le problème de Makespan (ou « répartition de tâches ») consiste à répartir différentes tâches sur plusieurs machines dans le but de réaliser toutes les tâches dans le plus court intervalle de temps. Chaque machine est considérée identique, aucune ne fonctionne plus vite qu’une autre ou n’est meilleure dans une certaine tâche.

Ce problème est NP-dur, il ne nous est donc impossible de réaliser un algorithme de complexité polynômiale pour un résultat exact (si P ≠ NP). Le but sera d’implémenter certains algorithmes d’approximation, puis de proposer le nôtre.

## Algorithmes

Nous avons implémenté 3 algorithmes :

* LSA (*List Simulated Annealing*)
* LPT (*Longest Processing Time*)
* MyAlgo (*Vente-Hartley*)

Cela nous permettra d’étudier ces algorithmes, leur complexité, leur approximation, …

## Équipe

L’équipe se compose de VENTE Maxime et HARTLEY Marc

# Le programme

## Conception

Nous souhaitons un programme en console compatible en multi-plateforme, de plus le projet est court et peu de fichiers seront nécessaires. Nous avons donc choisi le langage Java que nous maitrisons tous deux.

## Interface

L’interface se fait dans un terminal.

Un point qu’il nous a semblé important est de permettre à l’utilisateur d’exécuter le programme de la façon qu’il préfère. Il est donc important d’avoir un menu principal pour diriger l’utilisateur, mais aussi la possibilité de lancer le programme avec des options pour avoir directement un résultat sans interaction.

## Structure de données

Le programme a besoin de plusieurs informations pour fonctionner :

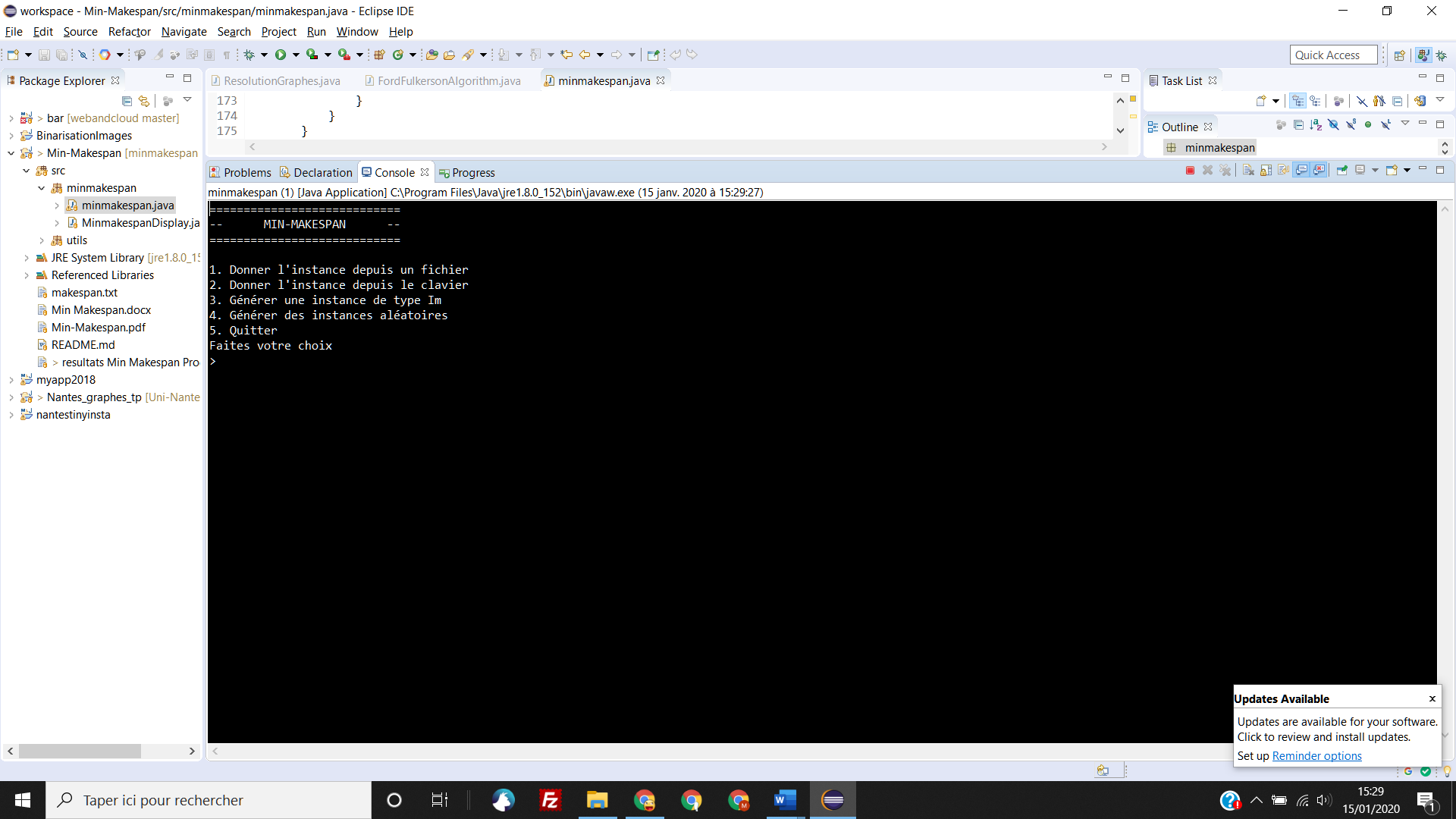
* Le nombre de machines *m*
* La durée des *n* tâches (qui peuvent être des entières ou décimales)

Ces données peuvent être renseignées de plusieurs manières :

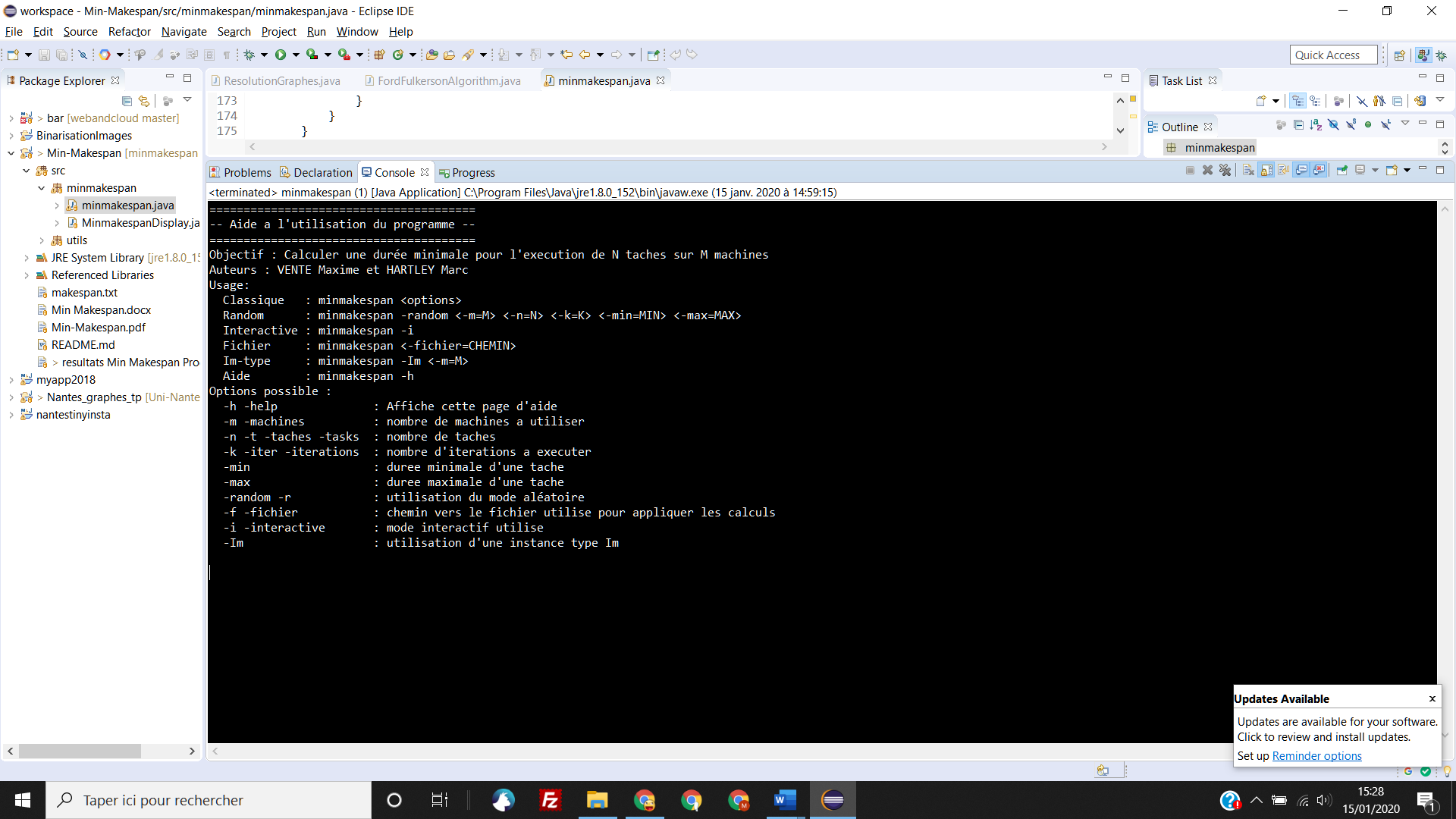
* Dans le mode « fichier » où l’utilisateur transmet le chemin du fichier qui sera utilisé, dans lequel il y a une unique ligne de la forme « *m:n:d1: d2: d3:… :dn-2:dn-1:dn »*avec *m* le nombre de machines, *n* le nombre de tâches à réaliser et *d1,d2…* la durée des tâches.
* Dans le mode « clavier » où l’utilisateur renseigne une ligne de la forme « *m:n:d1: d2: d3:… :dn-2:dn-1:dn »*avec *m* le nombre de machines, *n* le nombre de tâches à réaliser et *d1,d2…* la durée des tâches.
* Dans le type « Im » où l’utilisateur ne renseigne que le nombre *m* de machines, puis le programme calcule la durée des 2*m*+1 taches.
* Dans le type « random » où l’utilisateur renseigne :
  + Le nombre de machines
  + Le nombre de tâches à exécuter
  + Les bornes supérieures et inférieures des tâches à réaliser

Le programme utilisera des durées aléatoires (entre les bornes inférieures et supérieures) pour les *n* tâches à répartir sur les *m* machines.

## Usage

La façon la plus facile d’utiliser le programme est d’accéder au répertoire de l’exécutable puis de le lancer comme tout programme. Cela ouvre un menu principal avec différents choix possibles pour renseigner les données demandées (voir partie précédente « Structure de données »).

Une seconde façon d’exécuter le programme est de renseigner certaines informations dans les options disponibles en ligne de commande.

Par exemple, pour exécuter les algorithmes avec les données contenues dans un fichier, il suffit de lancer la commande : java -fichier=chemin\_vers\_le\_fichier minmakespan

Pour connaitre toutes les options possibles, il suffit de lancer la commande java --help minmakespan

## Algorithmes implémentés

Tous les algorithmes sont classés 2-approximation, c’est-à-dire que le résultat est au maximum le double du temps optimal.

### LSA

LSA (*List Simulated Annealing*) est l’algorithme intuitif que nous utiliserions : il suffit de prendre chaque tâche dans l’ordre renseigné puis de la faire exécuter par la première machine disponible. Répéter jusqu’à ce qu’il n’y ait plus de tâche inachevée.

### LPT

LPT (*Longest Processing Time*) est une variante du LSA dans laquelle la liste des tâches est triée par ordre décroissant de leur durée.

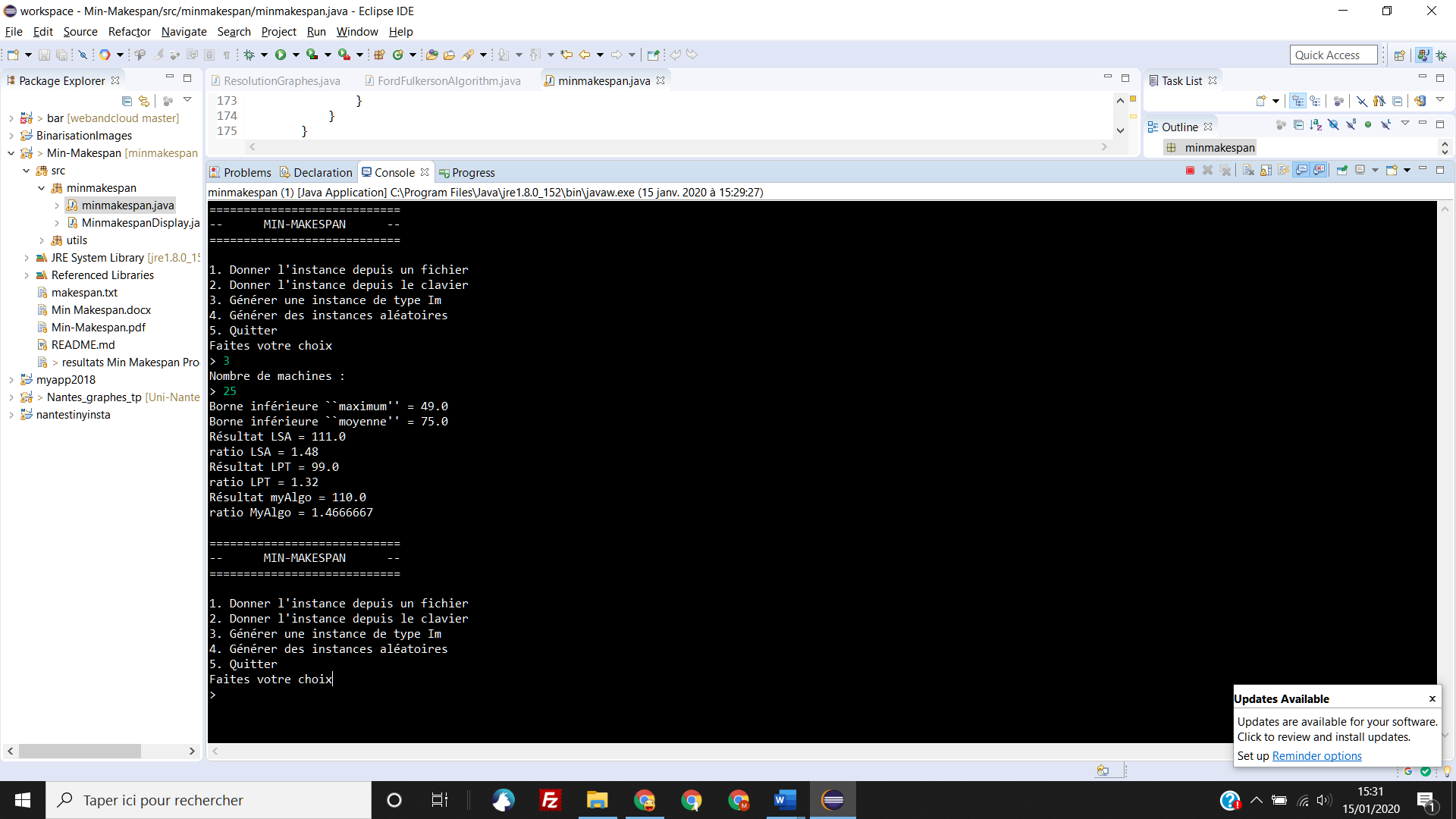
### MyAlgo

Cet algorithme mélange une variante de First-Fit avec le LPT.

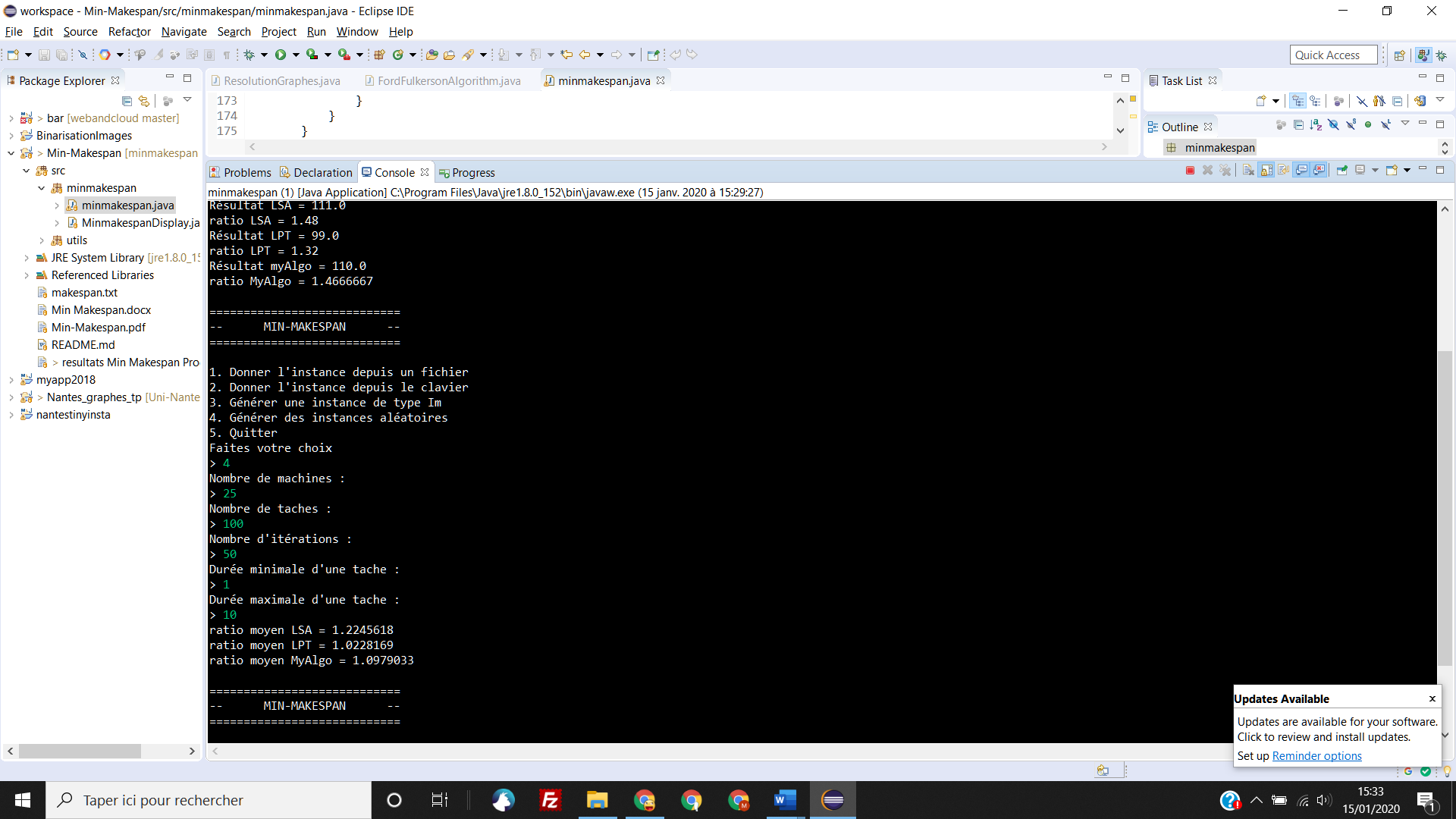
On suppose qu’on a *m* machines {M1; M2; … Mm}. On s’occupe tout d’abord de la machine M1 dont on affecte des tâches jusqu’à ce que cela lui prenne la moyenne des temps d’exécution par machine (), puis on continue l’opération sur M2, puis M3, et ainsi de suite jusqu’à Mn. Ensuite on réalise une LPT avec les tâches restantes et en partant du temps d’exécution actuel de chaque machine.

## Résultats du programme

Le programme retourne (sauf cas du mode « random ») les résultats de chaque algorithme avec le ratio calculé avec les bornes du temps optimal. Pour rappel, et



Avec le mode « random », on limite l’affichage à des « ratio moyens », c’est-à-dire les ratios calculés sur les *k* itérations demandées.



# Discussion sur le programme

## Complexité en temps

### Generation d’instances

#### Random

##### Pseudo-code :

Def doRandom(m : entier, n : entier, k : entier, min : décimal, max : décimal) :

Initialiser datas tableau 2-D de décimaux

Pour i allant de 1 à k :

Initialiser data tableau de décimaux

data[1] = m

data[2] = n

data[3 à 3 + n] = generation de n nombres décimaux aléatoires entre min et max

Ajouter data à datas

Fin Pour

Fin doRandom

##### Complexité :

On réalise une boucle k fois, dans laquelle on génère n nombre décimaux, on a donc une complexité de

#### Im

##### Pseudo-code :

Def doIm(m : entier) :

Initialiser data tableau de 2m+3 décimaux

data[1] = m

data[2] = n

durée = m

Pour i allant de 1 à 2n + 1 :

Si i > 3 et i est pair : ajouter 1 à durée

Ajouter durée à data

Fin Pour

Fin doIm

##### Complexité :

On a simplement une boucle de 2n+1 itérations, donc une complexité de

### Copie en mémoire des instances

#### If et Ic

##### Pseudo-code :

Def lectureFichier(fichier : String)

Récupérer dans ligne la première ligne du fichier fichier

Découper ligne à chaque « : »

m = ligne[1]

n = ligne[2]

initialiser un tableau de décimaux de taille n + 2, y ajouter m et n

Pour i allant de 1 à n :

Tableau[i + 2] = ligne[i + 2]

Fin Pour

Fin lectureFichier

Le code pour l’entrée de l’utilisateur est le même, mais au lieu de lire la première ligne du fichier, lire l’entrée utilisateur.

##### Complexité :

Une unique boucle allant de 1 à n, donc la complexité est de n.

### Algorithmes :

#### LSA :

##### Pseudo-code :

Def LSA(m : entier, n : entier, taches : tableau de décimaux, machines : tableau de décimaux)

Pour i allant de 1 à n :

Indice = indice de min(machines)

Machines[indice] += taches[i]

Fin Pour

Retourner machines

Fin LSA

Le paramètre machines permettra de réaliser une LSA après traitement dans MyAlgo.

##### Complexité :

On a une boucle de 1 à n, dans laquelle on cherche le min de machines. On a une complexité de soit .

#### LPT :

##### Pseudo-code :

Def LPT(m : entier, n : entier, taches : tableau de décimaux, machines : tableau de décimaux)

trier taches dans l’ordre décroissant

Réaliser LSA(m, n, taches, machines)

Fin LPT

##### Complexité :

On a une complexité de , soit . On peut supposer car le cas contraire signifie qu’il suffit de retourner le plus grand élément de taches (voir rapport des exercices, question 4). On a alors, pour conclure, une complexité

#### MyAlgo

##### Pseudo-code :

Def MyAlgo(m : entier, n : entier, taches : tableau de décimaux, machines : tableau de décimaux)

Moyenne = somme(taches)/m

TachesRestantes = tableau vide de décimaux

Pour iM allant de 1 à m :

Pour chaque tache T :

Si machines[iM] + T < moyenne :

Machines[iM] += T

Sinon :

Ajouter T à TachesRestantes

Fin Si

Retirer T de taches

Fin Pour

Fin Pour

Réaliser LPT(m, n, TachesRestantes, machines)

Fin MyAlgo

##### Complexité :

On a une complexité . On sait que contient 2 boucles imbriquées de m itérations puis n itérations et que , donc MyAlgo a une complexité. Comme vu dans la partie Complexité de LPT, on peut considérer cette complexité égale à . En appliquant la LSA plutôt que la LPT en fin d’algorithme, on peut réduire notre complexité à , sans beaucoup changer les résultats.

### Comparaison d’algorithmes

#### Sur Im

On a vu dans le rapport d’exercices (question 11 et 13) que LSA a un ratio d’approximation de 3/2 et LPT a un ratio de 4/3. MyAlgo a relativement les mêmes résultats que LPT, et tend à un ratio de 4/3 quand m est grand.

#### Sur Random

Pour étudier le degré d’approximation sur des listes de taches aléatoires, cela se complexifie car beaucoup de paramètres rentrent en jeu : m, n, min, max.

Pour m = 1000, n = 5000, min = 10, max = 100 :

LSA : 1.23 -- LPT : 1.03 – MyAlgo : 1.06

Pour m = 1000, n = 1500, min = 10, max = 100 :

LSA : 1.47 – LPT : 1.0 – MyAlgo = 1.12

Pour m = 1000, n = 5000, min = 1, max = 2

LSA : 1.15 – LPT : 1.01 – MyAlgo : 1.01

Pour m = 1000, n = 5000, min = 1, max = 100

LSA : 1.25 – LPT : 1.04 – MyAlgo : 1.07

On peut donc voir que rapprocher n de m dégrade le ratio de LSA et MyAlgo tandis que cela améliore celui de LPT.

De plus, augmenter le rapport max/min dégrade tous les ratios, mais MyAlgo en souffre plus que LPT.

### Classes d’instances « fétiches »

#### LSA

#### LPT

#### MyAlgo